Automated Verification of Concurrent Search Structures

Automated Verification of Concurrent Search Structures

Siddharth Krishna
Microsoft Research, Cambridge

Nisarg Patel
New York University

Dennis Shasha
New York University

Thomas Wies
New York University

SYNTHESIS LECTURES ON XYZ #13

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Search structures support the fundamental data storage primitives on key-value pairs: insert a pair, delete by key, search by key, and update the value associated with a key. Concurrent search structures are parallel algorithms to speed access to search structures on multicore and distributed servers. These sophisticated algorithms perform fine-grained synchronization between threads, making them notoriously difficult to design correctly. Indeed, bugs have been found both in actual implementations and in the designs proposed by experts in peer-reviewed publications. The rapid development and deployment of these concurrent algorithms has resulted in a rift between the algorithms that can be verified by the state-of-the-art techniques and those being developed and used today. The goal of this monograph is to bridge this gap and bring the certified safety of formal verification to the concurrent search structures used in practice. The techniques and frameworks we present can be applied to concurrent graph and network algorithms beyond search structures.

KEYWORDS

Verification, Separation Logic, Concurrency, Data Structures, Search Structures, B trees, Hash Structures, Log-Structured Merge Trees

Contents

Acknowledgments . . . . . **ix**

1 Introduction **1**
1.1 Algorithmic Modularity 2
1.2 Concurrent Search Structure Templates 4
1.3 Case Studies 5
1.4 Summary and Outline 6

2 The Edgeset Framework and Template Algorithms **8**
2.1 B-link Trees 8
2.2 Abstracting Search Structures using Edgesets 9
2.3 The Link Template 11

3 A Primer on Deductive Verification **12**
3.1 Basics and Notation 12
3.2 Programming Language 14
3.3 Separation Logic: Iris 16
3.3.1 Formulas 16
3.3.2 Specifications 19
3.3.3 Proof Rules 20

4 Ghost State **28**
4.1 Motivation 28
4.2 Ghost States and Resource Algebras 31
4.3 Proof of the Single-node Template 34
4.4 Two-node Template and Keysets 36
4.5 Disjoint Keysets and the Keyset RA 37

5 The Flow Framework **42**
5.1 Motivation 42
5.2 Local Reasoning about Global Properties 43
5.3 The Flow Interface RA 45

5.4 Encoding Keysets using Flows 47

6 Verifying Single-copy Concurrent Templates 50

6.1 The Give-up Template 51
6.1.1 Proof of the Give-up Template 52
6.1.2 Maintenance Operations. 57
6.2 The Link Template 58
6.2.1 Inreach 58
6.2.2 Proof of the Link Template 61
6.2.3 Maintenance Operations. 65
6.3 The Lock-coupling Template 66
6.4 Verifying Implementations 67
6.5 Proof Mechanization and Automation 67

7 Verifying Multicopy Structures 71

7.1 Overview 71
7.2 Differential File Structures 72
7.3 Log-Structured Merge Trees 74
7.4 Multicopy Structures 74

8 Verifying the Two-Node Multicopy Template 82

8.1 The Two-Node Multicopy Template 82
8.2 Correctness Proof for the Two-Node Template 83
8.2.1 Proving Search Recency 85
8.2.2 Proving the Correctness of Upsert 91
8.2.3 Proving the Correctness of Maintenance 94

9 Verifying a General Multicopy Template 100

9.1 The General Multicopy Template 100
9.2 Correctness Proof for the General Multicopy Template 103
9.2.1 Proving Search Recency 103
9.2.2 Proving the Correctness of Upsert 112
9.2.3 Proving the Correctness of Maintenance 114

10 Related Work, Future Work, and Conclusion 122

10.1 Related Work 122
10.2 Future Work 124

10.2.1 Generalizations and Extensions 124
10.2.2 Proving Liveness 124
10.2.3 Lock-Free Concurrent Search Structure Algorithms 125
10.3 Conclusion 126

Bibliography 128

Authors' Biographies 138

Acknowledgments

Besides acknowledging Liz, we should acknowledge our chosen readers Eddie ..., funding agencies, Microsoft, Iris and Grasshopper authors, and Diane Cerra and perhaps other folks at Morgan Claypool. Family members too if desired.

Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies
August 2020

CHAPTER 1

Introduction

For the last 60 years or so, the processing power of computers has been doubling approximately every two years. For most of that time, this growth has been backed by the increase in the number of transistors present on integrated circuit chips, a phenomenon commonly known as Moore's Law. Until about 2000, this was accompanied by a corresponding increase in clock speed, the speed at which computers perform each step of computation. However, in the early 2000s, computer hardware started reaching the physical limits of clock speed, mostly due to heating and quantum effects. To counter this, manufacturers have turned to parallel architectures where increased transistor densities are being used to provide multiple cores on a single chip, enabling multiple computations to be performed in parallel.

Having n processing cores on a chip, however, does not immediately imply a factor of n increase in speed. To make the most of these multicore machines, software needs to be carefully designed to efficiently divide work into threads, sequences of instructions that can be executed in parallel.

Well-designed parallel algorithms distribute the workload among threads in a way that minimizes the amount of time spent waiting for each other. A standard way to achieve this is to store any shared data in so-called concurrent data structures, supported by multi-threaded algorithms that store and organize data. These data structures are now core components of critical applications such as drive-by-wire controllers in cars, database algorithms managing financial, healthcare, and government data, and the software-defined-networks of internet service providers. The research and practitioner communities have developed concurrent data structure algorithms that are fast, scalable, and able to adapt to changing workloads.

Unfortunately, these algorithms are also among the most difficult software artifacts to develop correctly. Despite being designed and implemented by experts, the sheer complexity and subtlety of the ways in which different threads can interact with one another means that even these experts sometimes fail to anticipate subtle bugs. These bugs can cause the data to be corrupted or the program to misbehave in unexpected ways. For instance, consider the standard textbook on concurrent algorithms, *The Art of Multiprocessor Programming* [Herlihy and Shavit, 2008]. Although written by renowned experts who have developed many of the most widely used concurrent data structures, the errata of the book list several severe but subtle errors in the algorithms included in the book's first edition. There have also been many such examples of concurrent algorithms in peer-reviewed articles with (pencil-and-paper) mathematical proofs that have later turned out to contain mistakes [Burckhardt et al., 2007, Michael and Scott, 1995]. It is clear therefore that we desperately

need more systematic and dependable techniques to reason about and ensure correctness of these complex algorithms.

Formal verification is a field of research that aims to use mathematical techniques to prove, in a rigorous and machine-checkable manner, the absence of bugs and the conformity of a system to its intended specification. Several projects have demonstrated the successful use of formal verification to improve the reliability of real-world software designs, including SLAM, ASTREE, CertiKOS, seL4, CompCert, and Infer. In fact, in some areas such as hardware verification, formal verification is now a core part of the design process. However, there remains a huge gap between the concurrent data structures that can be verified by state-of-the-art techniques and the algorithms being developed and used everyday. The goal of this book is to bridge this gap and bring the certified safety of formal verification to the concurrent data structures in use everyday.

1.1 ALGORITHMIC MODULARITY

Concurrent data structures are among the most complex algorithms in use today. This is because they have to perform two very difficult tasks simultaneously: managing interference among threads in such a way as to ensure correctness, and organizing the data in memory so as to maximize performance. The resulting combination of delicate thread protocols and advanced data layouts used by concurrent data structures makes formally verifying them extremely challenging.

Modularity is as important in simplifying formal proofs as it has been for the design and maintenance of large systems. There are three main types of modular proof techniques: (i) Hoare logic [Hoare, 1969] enables proofs to be compositional in terms of program structure; (ii) separation logic [O'Hearn et al., 2001, Reynolds, 2002] allows proofs of programs to be local in terms of the state they modify; and (iii) thread modular techniques [Herlihy and Wing, 1990, Jones, 1983, Owicki and Gries, 1976] allow one to reason about each thread in isolation.

Concurrent separation logics [Brookes, 2007, Brookes and O'Hearn, 2016, da Rocha Pinto et al., 2014, Dinsdale-Young et al., 2013, 2010, Dodds et al., 2016, Feng et al., 2007, Fu et al., 2010, Jung et al., 2015, Nanevski et al., 2014, O'Hearn, 2007, Svendsen and Birkedal, 2014, Vafeiadis and Parkinson, 2007] incorporating all of the above techniques have led to great progress in the verification of practical concurrent data structures, including recent milestones such as a formal paper-based proof of the B-link tree [da Rocha Pinto et al., 2011].

An important reason why many proofs, such as that of the B-link tree, are still complicated is that they argue simultaneously about thread safety (i.e., how threads synchronize) and memory safety (i.e., how data is laid out in the heap). We contend that safety proofs should instead be decomposed so as to reason about these two aspects independently. When verifying thread safety we should abstract from the concrete heap structure used to represent the data and when verifying memory safety we should abstract from the concrete thread synchronization algorithm. Adding this form of abstraction as a fourth modular proof technique to our arsenal promises reusable proofs and simpler correctness arguments, which in turn aids proof automation.

As an example, consider the B-link tree, which uses the link-based technique for thread synchronization. The following analogy [Shasha and Goodman, 1988] captures the essence of the link-based technique on a macroscopic data structure.

Bob wants to borrow book k from the library. He looks at the library's catalog to locate k and makes his way to the appropriate shelf n. Before arriving at n, Bob gets caught up in a conversation with a friend. Meanwhile, Alice, who works at the library, reorganizes shelf n and moves k as well as some other books to n'. She updates the library catalog and also leaves a sticky note at n indicating the new location of the moved books. Finally, Bob continues his way to n, reads the note, proceeds to n', and takes out k. The synchronization protocol of leaving a note (the *link*) when books are moved ensures that Bob can find k.

The library patron corresponds to a thread searching for and performing an operation on the key k stored at some node n in the tree having links and the librarian corresponds to a thread performing a split operation involving nodes n and n'. As in our library analogy, the synchronization technique of creating a forward pointer (the link) when nodes are split works independently of how data is stored within each node and how these are organized in memory (e.g. as a B-tree or hash table). Hence, it applies to vastly different concrete data structures. Our goal is to verify the correctness of *template algorithms* once and for all so that their proofs can be reused across different data structure implementations.

Readers familiar with database protocols for transaction processing [Bernstein et al., 1987] will recall that the classical way to establish serializability of a set of concurrent transactions is to establish that transactions satisfy a stronger condition known as "conflict-preserving serializability." Conflict-preserving serializability requires that transaction operations (reads and writes) can be safely reordered to arrive at a serial execution. But no reordering is possible for Bob and Alice in the library, because *in no serial execution would Bob visit two shelves*. Thus, the execution of Bob and Alice are serializable (and, as we will see, "linearizable") but not conflict-preserving serializable, rendering the classical proof technique of transaction processing inapplicable.

The path to achieving *algorithmic proof modularity* is to combine the template abstractions with the proof technique of reasoning locally about modifications to the heap as in separation logic (SL). This combination leads to simple proofs that are easy to mechanize. In terms of the state of the art in verification technology, the proof of the link technique depends on certain invariants about the paths that a search for a key k follows in the data structure graph. However, with the standard heap abstractions used in separation logic (e.g. inductive predicates), it is hard to express these synchronization invariants independently of the invariants that capture how the data structure is represented in memory. That is why existing proofs such as the one of the B-link tree in [da Rocha Pinto et al., 2011] intertwine the synchronization invariants and the memory invariants, which makes the proof complex, hard to mechanize, and difficult to reuse on different structures.

1.2 CONCURRENT SEARCH STRUCTURE TEMPLATES

This book shows how to adapt and combine recent advances in compositional abstractions and separation logic in order to achieve the envisioned algorithmic proof modularity for the important class of *concurrent search structures*.

A search structure is a key-based store that implements three basic operations: search, insert, and delete. We refer to a thread seeking to search for, insert, or delete a key k as an operation on k, and to k as the *operation key*. We call the set of all keys the *key space* (e.g. the set of all natural numbers), written KS. For simplicity, the presentation in this book treats search structures as containing only keys (i.e. as implementations of mathematical sets), but all our proofs can be easily extended to consider search structures that store key-value pairs.

Our proof methodology for search structures is based on the template algorithms for concurrent search structures by Shasha and Goodman [1988], who identified the key invariants needed for decoupling reasoning about synchronization and memory representation for such data structures. We apply those templates here for single-copy structures (structures containing at most one copy of a key at any point in time, for example B-trees), and extend them significantly to cover multicopy algorithms (structures that may contain multiple copies of a single key, of which only the latest copy is active, for example log-structured merge trees).

The second ingredient is the concurrent separation logic Iris [Jung et al., 2016, 2018, 2015, Krebbers et al., 2017]. We show how to capture the high-level idea of [Shasha and Goodman, 1988] in terms of a new Iris resource algebra, yielding a general methodology for modular verification of concurrent search structures.

This methodology independently verifies (1) that the template algorithm satisfies the (atomic) abstract specification of search structures assuming that node-level operations maintain certain shape-agnostic invariants and (2) that the implementations of these operations for each concrete data structure maintains these invariants.

Our new resource algebra, in combination with Iris' notion of *atomic triples* [da Rocha Pinto et al., 2014, Jung et al., 2020, 2015], avoids explicit reasoning about execution histories and low-level programming language semantics. Moreover, it yields a local proof technique that eliminates the need to reason explicitly about the global abstract state of the data structure. The latter crucially relies on the recently proposed *flow framework* [Krishna et al., 2018, 2020b], the final ingredient of our methodology.

The flow framework, as explained in Chapter 5, provides an SL-based abstraction mechanism that allows one to reason about global inductive invariants of general graphs in a local manner. Using this framework, we can do SL-style reasoning about the correctness of a concurrent search structure template while abstracting from the specific low-level heap representation of the underlying data structure.

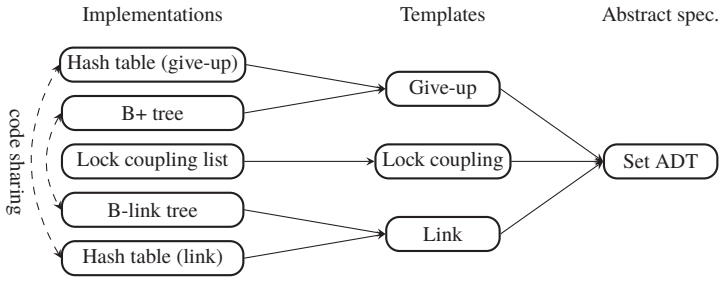

Figure 1.1: The decomposition of our proofs as a result of our template-based methodology. This figure shows our single-copy case studies (Chapter 6).

1.3 CASE STUDIES

We demonstrate our methodology by mechanizing the correctness proofs of template algorithms for single-copy (Chapter 6) and multicopy (Chapter 7) search structures that abstract common real-world data structure implementations.

The single-copy templates we verify are based on the link, give-up, and lock-coupling technique of synchronization (Figure 1.1). For these, we derive concrete verified implementations based on B-trees, hash tables, and sorted linked lists, resulting in five different data structure implementations.

In the second half of the book, we apply the same approach to concurrent multicopy search structures. We define and verify a two-node template that generalizes implementations such as differential file structures (Chapter 8). We also verify a more general template for multiple nodes that covers structures such as the log-structured merge (LSM) tree (Chapter 9).

A major advantage of our approach is that we can perform *sequential* reasoning when we verify that an implementation is a valid template instantiation. We therefore perform the template proofs in Iris/Coq and verify the implementations using the automated deductive verification tool GRASShopper [Piskac et al., 2013, 2014]. The automation provided by GRASShopper enables us to bring the proofs of highly complicated implementations such as B-link trees within reach.

Our proofs include a mechanization of the meta-theory of the flow framework presented in [Krishna et al., 2020b], carried out independently in both GRASShopper and Iris/Coq. The verification efforts in the two systems are hence each fully self-contained. The template proofs done in Iris are parametric with respect to any possible correct implementation of the node-level operations. The specifications assumed in Iris match those proved in GRASShopper. However, we note that there is no formal connection between the proofs done in the two systems. If one desires end-to-end certified implementations, one can perform both template and implementation proofs in Iris/Coq (albeit with substantial additional effort). Performing the proofs completely in GRASShopper or a similar SMT-based verification tool would require additional tooling effort to support reasoning about Iris-style resource algebras.

The proofs we obtain are more modular, reusable, and simpler than existing proofs of such concurrent data structures. For example, we are the first to obtain a mechanically verified proof of concurrent B-link trees, B+ trees based on the give-up technique, and log-structured merge trees. Our experience is that adapting our technique to a new template algorithm and instantiating a template to a new data structure takes only a few hours of proof effort.

1.4 SUMMARY AND OUTLINE

This book describes a template-based methodology for verifying concurrent search structures that enables proofs to be compositional in terms of program structure and state, and exploit thread and algorithmic modularity.

- Our methodology is based on the edgeset framework [Shasha and Goodman, 1988], which we describe in Chapter 2.
- All our proofs are performed in separation logic, in particular the higher-order concurrent separation logic Iris [Jung et al., 2018, 2015], which we introduce in Chapter 3.
- In Chapter 4, we formalize the reasoning of Shasha and Goodman [1988], by defining a novel resource algebra that allows us to use *ghost state* to keep track of *keysets*. This is a quantity that lets us separate the correctness of a search structure implementation from its concrete heap layout.
- Unfortunately, the keyset is a global graph property, so in order to obtain proof modularity we use the flow framework (presented in Chapter 5) to reason locally about modifications to the keyset.
- We demonstrate our proof technique on single-copy search structures in Chapter 6. We mechanically prove several complex real-world single-copy data structures such as the B-link tree and various hash structures.
- We turn our attention to multicopy search structures in Chapter 7. Chapter 8 presents a two-node template that is suitable for structures such as differential file systems. We extend this in Chapter 9 to more than two nodes, and verify a template that generalizes the LSM tree.
- In Chapter 10 we survey related work, discuss avenues for future work, and conclude.

Some of our work presented above, in particular the case studies presented in Chapter 6, has appeared in preliminary form in a conference paper [Krishna et al., 2020a].

Our proof technique that applies to any data structure that is indexed by keys, including implementations of sets, maps, and multisets (but, as of now, not other structures, e.g., queues and stacks). Our approach of separating concurrency templates and heap implementations requires the data structure to have an abstract state (e.g. as mathematical set or map) with a certain algebraic

structure: we need to be able to decompose the abstract state into local abstract states that are *disjoint* in some sense. Moreover, composition of abstract states needs to be associative, commutative, and homomorphic to composition of heap graphs. For instance, consider a binary search tree representing a mathematical map where each tree node stores a single key/value pair. If one arbitrarily splits the tree's heap graph into disjoint subgraphs, then these subgraphs represent disjoint mathematical maps whose union yields the map represented by the original composed heap graph. All search structure implementations that we know of satisfy these compositional properties.

Our case studies also include a mechanization of the meta-theory of the flow framework [Krishna et al., 2020b] within Coq and GRASShopper, as well as a construction of a flow-interface resource algebra for proofs in Iris. These components can be reused for other verification efforts such as communication network graphs and memory management.

In summary, this book follows in the tradition of simplifying and scaling up verification efforts using abstraction, compositionality, and modularity. We hope our framework for verifying concurrent search structures can serve as an inspiration for the design and proof of many algorithms for high-performance multi-threaded systems.

CHAPTER 2

The Edgeset Framework and Template Algorithms

This chapter introduces the template-based proof methodology used throughout this book. We first describe the B-link tree data structure, a highly-efficient and popular algorithm that uses the *link* technique of synchronization. We then introduce the notion of *edgesets* and show how they can be used to derive a template algorithm that can be instantiated to any search structure that uses the link technique, including the B-link tree. Edgesets, and template algorithms based on them, were first introduced by Shasha and Goodman [1988].

2.1 B-LINK TREES

The B-link tree (Figure 2.1) is an implementation of a concurrent search structure based on the B-tree. A B-tree is a generalization of a binary search tree, in that a node can have more than two children. In a binary search tree, each node contains a key k_0 and up to two pointers y_l and y_r. An operation on k takes the left branch if $k < k_0$ and the right branch otherwise. A B-tree generalizes this by having l sorted keys $k_0, \ldots, k_{l-1}$ and $l+1$ pointers $y_0, \ldots, y_l$ at each node, such that $B \leq l + 1 < 2B$ for some constant B. At internal nodes, an operation on k takes the branch y_i if $k_{i-1} \leq k < k_i$. In the most common implementations of B-trees (called B+ trees), the keys are stored only in leaf nodes; internal nodes contain "separator" keys for the purpose of routing only. When an operation arrives at a leaf node n, it proceeds to insert, delete, or search for its operation key in the keys of n. To avoid interference, each node has a lock that must be held by an operation before it reads from or writes to the node.

When a node n becomes full, a separate maintenance thread performs a split operation by transferring half its keys (and pointers, if it is an internal node) into a new node n', and adding a link to n' from the parent of n. A concurrent algorithm needs to ensure that this operation does not cause concurrent operations at n looking for a key k that was transferred to n' to conclude that k is not in the structure. The B-link tree solves this problem by linking n to n' and storing a key k' (the key in the gray box in the figure) that indicates to concurrent operations that the key k can be reached by following the link edge if $k > k'$. To reduce the time the parent node is locked, this split is performed in two steps: (i) a half-split step that locks n, transfers half the keys to n', and adds a link from n to n' and (ii) a complete-split performed by a separate thread that takes half-split nodes n, locks the parent of n, and adds a pointer to n'.

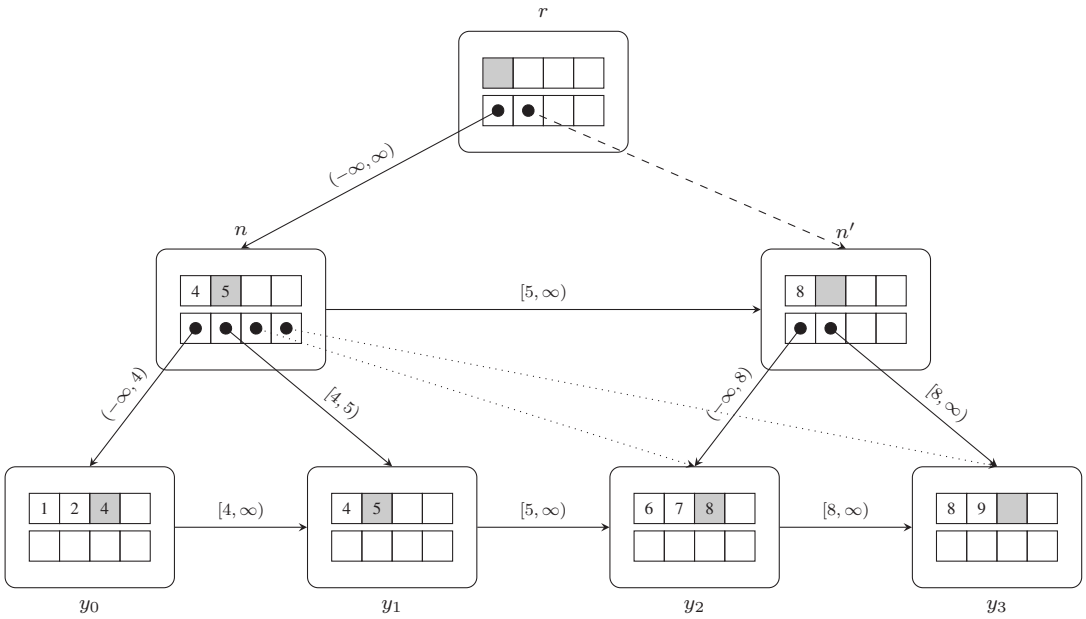

Figure 2.1: An example B-link tree state in the middle of a split. Node n was full, and has been half-split and children y_2 and y_3 have been transferred to the new node n' (old edges are shown with dotted lines), but the complete-split has yet to add n' to the parent r (the dashed edge). Each node contains an array of keys $k_0, \ldots, k_{l-1}$ in the top and an array of pointers $y_0, \ldots, y_l$ in the bottom. (The key in the gray box is not considered part of the contents and determines when to take the link edge.) Each edge is labelled by its edgeset (§2.2). Keys shown in internal nodes are separator keys, and do not count towards the contents of the structure. The search structure depicted here has contents $\{1, 2, 4, 6, 7, 8, 9\}$.

Figure 2.1 shows the state of a B-link tree where node y_2 has been fully split, and its parent n has been half split. The full split of y_2 moved keys $\{8, 9\}$ to a new node y_3, added a link edge, and added a pointer to y_3 in its (old) parent n. However, this caused n to become full, resulting in a half split that moved its children $\{y_2, y_3\}$ to a new node n' and added a link edge to n'. The key 5 in the gray box in n directs operations on keys $k \geq 5$ via the link edge to n'. The figure shows the state after this half split but before the complete-split when the pointer of n' will be added to r.

2.2 ABSTRACTING SEARCH STRUCTURES USING EDGESETS

The link technique is not restricted to B-trees: consider a hash table implemented as an array of pointers, where the ith entry refers to a bucket node that contains an array of keys $k_0, \ldots, k_l$ that all hash to i. When a node n gets full, it is locked, its keys are moved to a new node n' with twice the

```
let rec traverse n k =
  lockNode n;
  match findNext n k with
  | None -> n
  | Some n' ->
      unlockNode n;
      traverse n' k

let rec cssOp ω r k =
  let n = traverse r k in
  match decisiveOp ω n k with
  | None -> unlockNode n;
      cssOp ω r k
  | Some res -> unlockNode n;
      res
```

Figure 2.2: The link template algorithm. The `cssOp` method is the main method, and represents the core search structure operations (search, insert, and delete) via the parameter ω. It uses an auxiliary method `traverse` that recursively traverses the search structure until it finds the node upon which to operate. This template can be instantiated to the B-link tree algorithm by providing implementations of helper functions `findNext` and `decisiveOp`. `findNext` $n\,k$ returns `Some` n' if $k \in \mathsf{es}(n, n')$ and `None` if there exists no such n'. `decisiveOp` $n\,k$ performs the operation ω (either search, insert, or delete) on k at node n.

capacity, and n is linked to n'. Again, a separate operation locks the main array entry and updates it from n to n'.

While these two data structures look completely different, the main operations of search, insert, and delete follow the same abstract algorithm. In both, there is some local rule by which operations are routed from one node to the next, and both introduce link edges when keys are moved to ensure that no other operation loses its way.

To concretize this intuition, we view the state of a search structure abstractly as a mathematical graph. Each node in this graph can represent anything from two adjacent heap cells (in the case of a singly-linked list) to a collection of arrays and fields (in the case of a B-tree), and this mapping is determined by the specific implementation under consideration. We then define the *edgeset* of an edge (n, n'), written $\mathsf{es}(n, n')$, to be the set of operation keys for which an operation arriving at a node n traverses (n, n'). The B-link tree in Figure 2.1 labels each edge with its edgeset; the edgeset of (n, y_1) is $[4, 5)$ and the edgeset of the link edge (y_0, y_1) is $[4, \infty)$. Note that 4 is in the edgeset of (y_0, y_1) even though an operation on 4 would not normally reach y_0. This is deliberate. In order to make edgeset a local quantity, we say $k \in \mathsf{es}(n, n')$ if an operation on k would traverse (n, n') assuming it somehow found itself at n. In the hash table, assuming there exists a global root node, the edgeset from the root to the ith array entry is $\{k \mid hash(k) = i\}$, i.e., all the key values for which a search would go to the node of the ith array entry. By contrast, the edgeset from an array entry to the bucket node is the set of all keys KS, as is the edgeset from a deleted bucket node to its replacement. (The reason is that once we arrive at an array entry (or a deleted node), we follow the outgoing edge no matter what key we are looking for.)

2.3 THE LINK TEMPLATE

Figure 2.2 lists the link template algorithm [Shasha and Goodman, 1988] that uses edgesets to describe the algorithm used by all core operations for both B-link trees and hash tables in a uniform manner. The algorithm is described in an ML-like language that we use throughout the book, and is described in more detail in §3.2. The algorithm assumes that an implementation provides certain primitives or helper functions, such as `findNext` that finds the next node to visit given a current node n and an operation key k, by looking for an edge (n, n') with $k \in \mathsf{es}(n, n')$. For the B-link tree, `findNext` does a binary search on the keys in a node to find the appropriate pointer to follow. For the hash table, when at the root `findNext` returns the edge to the array element indexed by the hash of the key, and at bucket nodes it follows the link edge if it exists. The function `cssOp` can be used to build implementations of all three search structure operations by implementing the helper function `decisiveOp` to perform the desired operation (read, add, or remove) of key k on the node n.

An operation on key k starts at the root r, and calls a function `traverse` on line 9 to find the node on which it should operate. `traverse` is a recursive function that works by following edges whose edgesets contain k (using the helper function `findNext` on line 3) until the operation reaches a node n with no outgoing edge having an edgeset containing k. Note that the operation locks a node only during the call to `findNext`, and holds no locks when moving between nodes. `traverse` terminates when `findNext` does not find any n' such that $k \in \mathsf{es}(n, n')$, which indicates that n is the correct node to operate on. In the B-link tree example, this corresponds to finding the appropriate leaf.

At this point, the thread performs the decisive operation on n (line 10). If the operation succeeds, then `decisiveOp` returns `Some res` and the algorithm unlocks n and returns `res`. In case of failure (say an insert operation encountered a full node), the algorithm unlocks n, gives up, and starts from the root again.

If we can verify this link template algorithm with a proof that is parametrized by the helper functions, then we can reuse the proof across diverse implementations. In this book, we develop a proof technique that allows us to verify such template algorithms for a wide variety of search structures, from simple give-up based hash tables to cutting-edge implementations of log-structured merge trees.

CHAPTER 3

A Primer on Deductive Verification

In this chapter, we introduce and illustrate the formal techniques used in this book, using an extremely simple search structure template and implementation as an example.

3.1 BASICS AND NOTATION

We begin with some basic definitions and notation.

- The term $(b \mathbin{?} t_1 : t_2)$ denotes t_1 if condition b holds and t_2 otherwise.
- We write $f \colon A \to B$ for a function from A to B, and $f \colon A \rightharpoonup B$ for a partial function from A to B.
- For a partial function f, we write $f(x) = \bot$ if f is undefined at x.
- We use the lambda notation $(\lambda x.\ E)$ to denote a function that maps x to the expression E (typically containing x).
- If f is a function from A to B, we write $f[x \rightarrowtail y]$ to denote the function from $A \cup \{x\}$ defined by $f[x \rightarrowtail y](z) \coloneqq (z = x \mathbin{?} y : f(z))$.
- We use $\{x_1 \rightarrowtail y_1, \ldots, x_n \rightarrowtail y_n\}$ for pairwise different x_i to denote the function $\epsilon[x_1 \rightarrowtail y_1] \cdots [x_n \rightarrowtail y_n]$, where ϵ is the function on an empty domain.
- Given functions $f_1 \colon A_1 \to B$ and $f_2 \colon A_2 \to B$ we write $f_1 \uplus f_2$ for the function $f \colon A_1 \uplus A_2 \to B$ that maps $x \in A_1$ to $f_1(x)$ and $x \in A_2$ to $f_2(x)$ (if A_1 and A_2 are not disjoint sets, $f_1 \uplus f_2$ is undefined).
- We also write $\lambda_0 \coloneqq (\lambda m.\ 0)$ for the identically zero function, $\lambda_{\mathsf{id}} \coloneqq (\lambda m.\ m)$ for the identity function, and use $e \equiv e'$ to denote function equality.
- For functions f_1, f_2, we write $f_2 \circ f_1$ to denote function composition, i.e. $(f_2 \circ f_1)(x) = f_2(f_1(x))$, and use superscript notation f^p to denote the function composition of f with itself p times.

- For multisets, we use the standard set notation if it is clear from the context. We also write $\{x_1 \rightarrowtail i_1, \ldots, x_n \rightarrowtail i_n\}$ for the multiset containing i_1 occurrences of x_1, i_2 occurrences of x_2, etc. For a multiset S, we write $S(x)$ to denote the number of occurrences of x in S.

- We write _ for an anonymous variable, usually when a variable is used only once.

We now turn to introducing some basic algebraic concepts.

Definition 3.1 A *partial monoid* is a set M, along with a partial binary operation $+\colon M \times M \rightharpoonup M$, and a special zero element $0 \in M$, such that (1) $+$ is associative, i.e., $(m_1 + m_2) + m_3 = m_1 + (m_2 + m_3)$; and (2) 0 is an identity element, i.e., $m + 0 = 0 + m = m$. Here, equality means that either both sides are defined and equal, or both sides are undefined.

Partial monoids are the basis of ghost state, an important reasoning technique which will be introduced in Chapter 4. An example of a partial monoid is the set $\mathcal{P}(\mathbb{N})$ of all finite subsets of natural numbers, together with disjoint union $\uplus$ (where $X_1 \uplus X_2$ is undefined if they are not disjoint) and the empty set $\emptyset$. We usually identify a partial monoid with its support set M.

If $+$ is a total function, then we call M a monoid. For example, the set of natural numbers $\mathbb{N}$ together with addition $+$ and zero, form a monoid. Let $m_1, m_2, m_3 \in M$ be arbitrary elements of the (partial) monoid in the following. Here is some terminology and notation associated with (partial) monoids:

- We call a (partial) monoid M *commutative* if $+$ is commutative, i.e., $m_1 + m_2 = m_2 + m_1$. $\mathbb{N}$ and $\mathbb{Z}$ are commutative monoids, while 2×2 matrices of natural numbers with matrix multiplication and the identity matrix form a non-commutative monoid.

- Similarly, a commutative (partial) monoid M is *cancellative* if $+$ is cancellative, i.e., if $m_1 + m_2 = m_1 + m_3$ is defined, then $m_2 = m_3$. For example, $\mathbb{N}$ is a cancellative while the monoid formed by sets of natural numbers under set union is not (as $\{1\} \cup \{1, 2\} = \{1\} \cup \{2\}$).

- We say M is *positive* if $m_1 + m_2 = 0$ implies that $m_1 = m_2 = 0$. As you might expect, $\mathbb{N}$ is a positive partial monoid, while $\mathbb{Z}$ is not positive.

- For a positive (partial) monoid M, we can define a partial order $\leq$ on its elements as $m_1 \leq m_2$ if and only if $\exists m_3.\ m_1 + m_3 = m_2$. Positivity also implies that every $m \in M$ satisfies $0 \leq m$. For $\mathbb{N}$, this order corresponds to the natural less-than-or-equal-to ordering on natural numbers.

We will see commutative cancellative monoids used in Chapter 5 in order to set up the flow framework for reasoning locally about global graph properties.

3.2 PROGRAMMING LANGUAGE

The programming language that we use in this book is an ML-like language with higher-order store, fork, and compare-and-set (**CAS**), whose grammar is given below:

$v \in \mathsf{Val} ::=$	`()`	Unit
$\mid$	z	Integers
$\mid$	`true` $\mid$ `false`	Booleans
$\mid$	ℓ	Heap locations
$\mid$	$(\mu\ f\ x.\ e)$	Fixpoints
$e \in \mathsf{Expr} ::=$	v	Value
$\mid$	x	Variable
$\mid$	$e_1\ e_2$	Function application
$\mid$	**ref**(e)	Reference creation
$\mid$	$!e$	Dereference
$\mid$	$e_1 \leftarrow e_2$	Reference assignment
$\mid$	**CAS**(e, e_1, e_2)	Compare-and-set
$\mid$	**fork** $\{e\}$	Fork
$\mid$	**let** $x = e_1$ **in** e_2	Let expression
$\mid$	**if** e_1 **then** e_2 **else** e_3	Conditional expression

Values in our language include the unit value (the only value of the unit type), integers, booleans, heap locations or addresses, and fixpoints $(\mu\ f\ x.\ e)$ that describes the function that is the least fixpoint of the equation $f(x) = e$. Lambda abstractions $(\lambda x.\ e)$, that describe a function that maps argument x to the expression e, can be defined in terms of fixpoints as $(\lambda x.\ e) \coloneqq (\mu\ \_\ x.\ e)$.

Since this is a functional language, all programs are expressions, the simplest kind of which is just a value or a variable. Function application is written in standard functional programming style as `foo arg` instead of `foo(arg)`. The reference creation expression **ref**(e) evaluates to the address of a newly allocated heap location, whose value is set to the result of evaluating the expression e. Heap locations, or references, can be read (or *dereferenced*) by using the $!e$ command, where e evaluates to a heap location. A reference at location e_1 can be written to value e_2 using the $e_1 \leftarrow e_2$ command, which returns the unit value `()`. The compare-and-set command **CAS**(e, e_1, e_2) is an instruction that checks if the value at heap location e is equal to e_1; if so, it sets its value to e_2 and returns `true`, otherwise it does nothing and returns `false`. The last three operations are *atomic*, which means that during their execution no other thread can read or write to the heap location they operate on (in other words, they appear to take place instantaneously). The fork command **fork** $\{e\}$ creates a new thread that evaluates expression e, and returns the unit value `()`. We have the standard compound expressions such as let-bindings and if-then-else expressions. In the rest of the book, we use standard syntactic shorthands, such as:

$e_1;\ e_2 \coloneqq$ **let** $\_ = e_1$ **in** e_2 $\qquad$ **let rec** $f x$ `=` $e1$ **in** $e2 \coloneqq$ **let** f `=` $(\mu\ f\ x.\ e1)$ **in** $e2$

```
1  let rec cssOp ω r k =
2    lockNode r;
3    let res = decisiveOp ω r k in
4    unlockNode r;
5    res
6
7  let lockNode x =
8    if CAS(lk(x), false, true) then
9      ()
10   else
11     lockNode x
12
13 let unlockNode x =
14   lk(x) ← false
```

```
1  let decisiveOp ω r k =
2    match ω with
3    | search -> search r k
4    | insert -> insert r k
5    | delete -> delete r k
6
7  let insert r k =
8    let u = !r in
9    if u == ⊥ then
10     r ← k;
11     true
12   else
13     insert r k
```

Figure 3.1: A template algorithm for a single-node search structure (left), and an extremely simplistic example of an implementation for the single-node template (right).

For example, consider the program on the left of Figure 3.1. This is a template algorithm for the simplest possible search structure, a single-node structure, which we use as a running example in this chapter. As with all the template algorithms in this book, we have a method `cssOp` that stands for any one of the three core operations, by means of the parameter ω (which is one of search, insert, or delete). `cssOp` first locks the (only) node r, then calls a function `decisiveOp` on the locked node, before unlocking the node and returning the result. For simplicity, we use a spinlock to implement `lockNode` and `unlockNode`. We also assume that there exists a function $\mathsf{lk}(x)$ that maps each node x to the heap address that it uses as a lock flag (true when locked, false otherwise). x is locked by repeatedly trying to **CAS** $\mathsf{lk}(x)$ to true, which will succeed only if the node is unlocked. Unlocking is simpler, and the thread that has locked a node sets $\mathsf{lk}(x)$ to false. We call this algorithm a template algorithm because it does not specify the implementation of its helper functions, in this case, only the single function `decisiveOp`. This operation depends on the concrete implementation of the data structure: on how nodes are laid out in memory, and how keys are stored within nodes.

Figure 3.1 (right) also shows a simple implementation of the single-node template. The simplest implementation we can think of is for the single node r to consist of a single heap location (also, for convenience, at address r). Our implementation of insert reads the address r into variable u and if u is empty (denoted by some default initial value $\perp$), writes the operation key k to location r. However, if u is not empty, then this simplistic implementation loops infinitely. A slightly more realistic implementation could, for example, have the single node contain an array of keys, which is dynamically resized when full.

3.3 SEPARATION LOGIC: IRIS

In this book, we use the de facto standard way to specify and verify data structures: *separation logic*. Separation logic (SL), is an extension of Hoare logic [Hoare, 1969] that is tailored to perform modular reasoning about programs that manipulate mutable resources. In other words, SL is a language that allows one to succinctly and modularly describe states of a program. Each sentence in this language is called a *formula*, and describes a set of states. Separation logic also gives us a set of proof rules that can be used to prove that states of interest (such as the set of resulting states after a program executes) satisfy a particular formula.

There are many incarnations of SL, each tailored to reasoning about a particular class of programs. In this book, we use *Iris*, a mechanized higher-order concurrent separation logic framework. The distinguishing feature of Iris is its generality: it is designed as a small set of core primitives and proof rules that can be used to encode a huge variety of common constructs and techniques for reasoning about concurrent programs. In particular, Iris is easily extendable with user-defined resources via its ghost state mechanism, which we will describe in Chapter 4.

The price paid for this generality is that the core Iris logic is very abstract. To make this book accessible to a wider audience, we present many of the derived features as though they are primitive Iris features, and avoid talking about the formal semantic model altogether. We refer the interested reader to a paper by Jung et al. [2018] for a more detailed introduction to Iris, and to the documentation [Iris Team, 2020] for the full details.

Note that Iris is a garbage-collected or *intuitionistic* separation logic, hence all programs in this book assume a garbage-collected setting. We can extend the techniques presented in this book to also prove absence of memory leaks by considering extension of Iris such as Iron [Bizjak et al., 2019].

3.3.1 FORMULAS

Iris formulas describe the *resources* owned by a thread. These resources can be part of a concrete program state, for example, a set of heap cells, which captures the situations in which a thread has exclusive ownership over these cells (because, say, it has locked them). In order to reason about fine-grained concurrency and more complex concurrent patterns, these resources can also capture shared ownership and partial knowledge of shared program state. In order to build intuition, we focus on the simple case where formulas describe concrete program states, in the form of subsets of the heap, and defer the discussion of advanced resources to Chapter 4. We say a state σ *satisfies* a formula when the state is described by the formula. For a formal definition of program states and the satisfaction relation in Iris, see the Iris documentation [Iris Team, 2020].

The grammar of Iris formulas is shown in Figure 3.2, and includes the following constructs:

- The first line consists of standard boolean constructs: boolean constants true and false, then conjunction, disjunction, and implication.

$$
\begin{aligned}
P, Q, R :=\ & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \\
& \mid \exists x.\ P \mid \forall x.\ P \\
& \mid x \mapsto v \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathop{\Large\ast}_{x \in X} P \\
& \mid \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} \mid \{P\}\ e\ \{v.\, Q\} \mid \langle\, P \,\rangle\ e\ \langle\, v.\, Q \,\rangle
\end{aligned}
$$

Figure 3.2: The grammar of Iris formulas, restricted to the constructs used in this book.

- The second line introduces quantification, although since Iris is a *higher-order* logic, x can range over any type, including that of propositions and (higher-order) predicates.

- The third line includes the *points-to* assertion $x \mapsto v$, a primitive assertion that denotes a heap cell at address x containing value v. Note that if a state σ satisfies $x \mapsto v$, then σ is a singleton heap containing only the heap address x.

- A powerful feature of SL is the $*$ connective, or *separating conjunction*, that is used to conjoin two *disjoint* parts of the heap. A state σ satisfies $P * Q$ if it can be broken up into two disjoint states $\sigma = \sigma_1 \circledcirc \sigma_2$ such that σ_1 satisfies P and σ_2 satisfies Q. When P and Q are formulas denoting heaps, this means that they talk about disjoint regions of the heap, i.e. that they do not have any heap addresses in common. In particular, this means that $x \mapsto v_1 * y \mapsto v_2$ implies that $x \neq y$, and the formula $x \mapsto y * x \mapsto z$ is unsatisfiable as it is not possible to split a heap into two disjoint portions both of which contain the same address x.

- We also have an iterated version of separating conjunction: $\mathop{\Large\ast}_{x \in X} P$, where the bound variable x ranges over a set X. For example, $\mathop{\Large\ast}_{x \in X} x \mapsto 13$ denotes a set of heap cells with address X all of whom have the value 13.

- The *separating implication* connective $\mathrel{-\!\!*}$, is defined as: a state σ satisfies $P \mathrel{-\!\!*} Q$ if for every state σ_1 disjoint from σ that satisfies P, the combined state $\sigma \circledcirc \sigma_1$ satisfies $P \mathrel{-\!\!*} Q$. The best way to understand $\mathrel{-\!\!*}$ is to think of $*$ and $\mathrel{-\!\!*}$ as separation logic analogues of $\wedge$ and $\Rightarrow$ from first-order logic. For instance, $P * Q$ means you have both P *and* Q (and that they are disjoint, or more generally, composable). Similarly, $P \mathrel{-\!\!*} Q$ describes a state such that if you conjoin it with P, then you get Q. This property, $P * (P \mathrel{-\!\!*} Q) \vdash Q$, is the SL analogoue of *modus ponens* in first-order logic ($P \wedge (P \Rightarrow Q) \vdash Q$).[1]

- Finally, the last line contains formulas for invariants (explained in Chapter 8), ghost state (explained in Chapter 4), and Hoare and atomic triples (explained in §3.3.2).

[1] $\vdash$ is the *provable entailment* relation; $P \vdash Q$ means if we can prove P is true, then we can prove that Q is true.

Returning to our single-node search structure example (Figure 3.1), we could describe the state of the structure by the formula

$$\exists v.\ r \mapsto v,$$

which means r is a heap location containing value v. Such a description would be quite limiting: we would allow implementations only where each node contains a single heap location, implying that the search structure would become full as soon as a key is added to it, and we would not be able to provide implementations for `lockNode` and `unlockNode` as most lock implementations require additional memory (e.g. a bit to hold the lock state). To solve the second issue, we could instead use the formula

$$\exists v, b.\ r \mapsto v * \mathsf{lk}(r) \mapsto b,$$

where the second heap location (at address $\mathsf{lk}(r)$) stores the lock bit. Note that the use of the separating conjunction $*$ implies that $r \neq \mathsf{lk}(r)$, and so this formula describes two distinct heap locations. If we wanted to describe an implementation where our single node contained an array of length $N+1$, we could use the formula

$$\exists v_0, v_1, \ldots, v_N, b.\ \ r \mapsto v_0 * (r+1) \mapsto v_1 * \cdots * (r+N) \mapsto v_N * \mathsf{lk}(r) \mapsto b,$$

which describes $N+1$ consecutive heap locations and a distinct lock location. This can also be expressed succinctly using the iterated separating conjunction:

$$\exists v_0, v_1, \ldots, v_N, b.\ \ \mathsf{lk}(r) \mapsto b\ *\ \mathop{\Large *}_{0 \le i \le N} (r+i) \mapsto v_i$$

Recall, however, that we want to leave the concrete details of how the node stores its keys to the implementation of the template algorithm. This means we need to use some kind of a template formula, one that can be instantiated to capture the instantiations described above. This can be achieved by using an *abstract predicate* $\mathsf{node}(n, C_n)$, that stands for an unspecified implementation of a node at address n containing keys C_n. Our template proof can then describe the state of the search structure by the formula

$$\exists b.\ \mathsf{node}(r, C) * \mathsf{lk}(r) \mapsto b,$$

which describes a state containing a single node r, with contents C, and a lock flag at location $\mathsf{lk}(r)$ that has some contents b.

Our template proof then specifies the assumptions it makes about the predicate. As long as implementations of node satisfy these assumptions, the template proof will be valid. For instance, all our template proofs assume that

$$\forall n, C_n, C'_n.\ \mathsf{node}(n, C_n) * \mathsf{node}(n, C'_n) \mathrel{-\!\!*} \mathsf{False},$$

an analogue to the property of points-to predicates ($x \mapsto y * x \mapsto z \mathrel{-\!\!*} \mathsf{False}$) that captures the intuition that the $\mathsf{node}(n, C_n)$ predicate "owns" the heap address n which cannot be in two places at once.

$$\Psi_\omega(k, C, C', \mathsf{res}) \coloneqq \begin{cases} C' = C \land (\mathsf{res} \iff k \in C) & \omega = \texttt{search} \\ C' = C \cup \{k\} \land (\mathsf{res} \iff k \notin C) & \omega = \texttt{insert} \\ C' = C \setminus \{k\} \land (\mathsf{res} \iff k \in C) & \omega = \texttt{delete} \end{cases}$$

Figure 3.3: The desired specification of core search structure operations. For core operation ω on key k, Ψ_ω describes the relation between C, the contents before the operation, C', the contents after the operation, and the return value res.

3.3.2 SPECIFICATIONS

We now turn to the question of how to specify a program using SL formulas. The basic form of program specification is the *Hoare triple* [Hoare, 1969] $\{P\}\ e\ \{v.\, Q\}$ which is true if for every state σ that satisfies P we have that (1) the program e does not reach an error state when run from σ (for example, by trying to read unallocated memory), and (2) that if e terminates then it returns some value v and the new state is some σ' that satisfies Q. We write $\{P\}\ e\ \{Q\}$ in the case where Q does not mention the return value v.

In the single-node template, we can use a Hoare triple to specify the behavior the template expects from the helper function `decisiveOp`:

$$\{\mathsf{node}(r, C)\}\ \texttt{decisiveOp}\ \omega\ \texttt{r}\ \texttt{k}\ \{\mathsf{res}.\, \mathsf{node}(r, C') * \Psi_\omega(k, C, C', \mathsf{res})\}$$

The predicate Ψ_ω, defined in Figure 3.3, captures the behavior of the set abstract data type, and is the abstract specification of search structures that we use throughout this book. For a given search structure operation ω on key k, $\Psi_\omega(k, C, C', \mathsf{res})$ relates the contents of the search structure before the operation (C), to the contents after the operation (C'), and the return value res. The Hoare triple for `decisiveOp` says: given a state containing a node r with contents C, the operation ω returns a value res and the node r has (new) contents C' such that Ψ_ω expresses the relationship between the old and new contents and res.

Our goal is to prove that the single-node template algorithm is a correct search structure. We cannot, however, use a similar Hoare triple to specify the single-node template. The reason is that Hoare triples are suitable for specifying sequential algorithms, but not very helpful when reasoning about concurrent algorithms. Imagine we tried to give the following specification for `cssOp`, where $\mathsf{CSS}(r, C)$ (for concurrent search structure) is a predicate describing a search structure at location r with contents C:

$$\{\mathsf{CSS}(r, C)\}\ \texttt{cssOp}\ \omega\ \texttt{r}\ \texttt{k}\ \{\mathsf{res}.\, \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', \mathsf{res})\}$$

Unfortunately, such a specification would rule out most concurrent search structure algorithms. The reason is that Hoare triples are defined in such a way that when proving a triple for a program, one can assume that any resources in the precondition are owned exclusively by the thread

executing the program. This means that any program that calls `cssOp` must own $\mathsf{CSS}(r, C)$ exclusively, meaning it can call it only in situations where no other thread can read or modify $\mathsf{CSS}(r, C)$. For example, consider a simple concurrent data structure protected by a single global lock. Before performing any operation on the structure, threads acquire this global lock, thereby ensuring no other thread can access the structure. Essentially, Hoare triples permit only coarse-grained, so-called disjoint concurrency.

We specify the concurrent behavior of search structures using *atomic triples* [da Rocha Pinto et al., 2014, Jung et al., 2020, 2015]. An atomic triple $\langle\, P \,\rangle\ e\ \langle\, v.\ Q \,\rangle$ is made up of the precondition P, postcondition Q, return value v, and a program e. Such a triple means that e, despite executing in potentially many atomic steps, appears to operate atomically on the shared state and transform it from a state satisfying P to one satisfying Q. Atomic triples are strongly related to the well-known *linearizability* [Filipovic et al., 2009, Herlihy and Wing, 1990] criterion for concurrent algorithms. Intuitively, there is a point in time during the course of the execution of e, known as the *linearization point*, where e updates P to Q. For the example of an insert operation on a search structure, this will be the point of time when the inserted value is visible to other threads. Linearizability requires that a concurrent set of operations produces the same final state and returns the same values as a sequential execution of the operations where the ordering is the order of the linearization points. In other literature [Bernstein et al., 1987], linearizability is known as order-preserving serializability.

The specification we want to prove for the single-node template algorithm is the following:

$$\langle\, C.\ \mathsf{CSS}(r, C) \,\rangle\ \texttt{cssOp}\ \omega\ rk\ \langle\, \mathsf{res}.\ \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', \mathsf{res}) \,\rangle \tag{3.1}$$

The binder on C in the precondition is a special *pseudo-quantifier* that captures the fact that during the execution of ω, the value of C can change (e.g. by concurrent operations). At the linearization point however, `cssOp` changes $\mathsf{CSS}(r, C)$ to $\mathsf{CSS}(r, C')$ in an atomic step. The new set of keys C' and the eventual return value res satisfy the predicate $\Psi_\omega(k, C, C', \mathsf{res})$. Note that the C in the postcondition is bound in the precondition, i.e. to the contents *just before* the linearization point. The goal is that clients of the search structure can pretend that they are using a serial or sequential implementation with specification Ψ_ω.

Let us turn now to the question of how to prove specifications of programs.

3.3.3 PROOF RULES

First, let us look at how to prove a Hoare triple. Figure 3.4 lists some of the standard proof rules useful for proving Hoare triples.[2] An inference rule consists of two parts separated by a horizontal line: the part above the line contains one or more premises, and the part below the line contains the conclusion. For example, the way to read the HOARE-FORK rule is that if we can prove that expression e satisfies the Hoare triple $\{P\}\ e\ \{\mathsf{True}\}$ then we can apply the rule and infer that e also satisfies the triple $\{P\}\ \texttt{fork}\ \{e\}\ \{\mathsf{True}\}$. Typically, when performing a proof we use the rule in a bottom-up fashion. We start with a specification that we want to prove (the *goal*), and then find an inference

[2] We write $e[x \rightarrowtail v]$ to denote the expression e after substituting all occurences of the variable x with the term v.

HOARE-RET
$$\{\mathsf{True}\}\ w\ \{v.\, v = w\}$$

HOARE-FALSE
$$\{\mathsf{False}\}\ e\ \{v.\, P\}$$

HOARE-ALLOC
$$\{\mathsf{True}\}\ \texttt{ref}(v)\ \{\ell.\, \ell \mapsto v\}$$

HOARE-LOAD
$$\{\ell \mapsto v\}\ !v\ \{w.\, \ell \mapsto v * w = v\}$$

HOARE-CAS-SUC
$$\{\ell \mapsto v\}\ \texttt{CAS}(\ell, v, w)\ \{b.\, b = \mathsf{True} * \ell \mapsto w\}$$

HOARE-STORE
$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{\ell \mapsto w\}$$

HOARE-CAS-FAIL
$$\frac{v \neq v'}{\{\ell \mapsto v\}\ \texttt{CAS}(\ell, v', w)\ \{b.\, b = \mathsf{False} * \ell \mapsto v\}}$$

HOARE-LAM
$$\frac{\{P\}\ e[x \rightarrowtail v]\ \{w.\, Q\}}{\{P\}\ (\lambda x.\, e)\ v\ \{w.\, Q\}}$$

HOARE-FORK
$$\frac{\{P\}\ e\ \{\mathsf{True}\}}{\{P\}\ \texttt{fork}\ \{e\}\ \{\mathsf{True}\}}$$

HOARE-LET
$$\frac{\{P\}\ e_1\ \{w.\, R\} \qquad \forall w.\, \{R\}\ e_2[x \rightarrowtail w]\ \{v.\, Q\}}{\{P\}\ \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \{v.\, Q\}}$$

HOARE-CSQ
$$\frac{P \Rightarrow P' \qquad \{P'\}\ e\ \{v.\, Q'\} \qquad \forall v.\, Q' \Rightarrow Q}{\{P\}\ e\ \{v.\, Q\}}$$

HOARE-FRAME
$$\frac{\{P\}\ e\ \{v.\, Q\}}{\{P * R\}\ e\ \{v.\, Q * R\}}$$

HOARE-DISJ
$$\frac{\{P\}\ e\ \{v.\, R\} \qquad \{Q\}\ e\ \{v.\, R\}}{\{P \vee Q\}\ e\ \{v.\, R\}}$$

HOARE-EXIST
$$\frac{\forall x.\, \{P\}\ e\ \{v.\, Q\}}{\{\exists x.\, P\}\ e\ \{v.\, Q\}}$$

Figure 3.4: Proof rules for establishing Hoare triples.

rule that matches the structure of the goal and apply it to replace the goal with the premises of the rule. A rule with no premises is called an *axiom*, and in this case we omit the horizontal line (e.g. HOARE-LOAD). Some rules are bi-directional: the premise implies the conclusion, but the conclusion also implies the premise. These rules are denoted with a double horizontal line (e.g. HOARE-EXIST).

A common metaphor used when describing SL proofs is that of ownership of resources. We introduced the formula $x \mapsto \_$ as describing a program state (a heap cell at address x), but going forward we will think of this points-to predicate as a *resource* owned by the program we are trying to verify. It is a resource because it allows us to safely perform a read or write to heap location x, using rules HOARE-LOAD and HOARE-STORE. When proving a Hoare triple, we obtain the resources in the precondition when we begin, and we are under the obligation to transform them into the resources in the postcondition by the end of the program.

Consider the following program expression that reads the value stored at heap location x into a variable ($v := !x$) and then writes $v + 1$ back into the location (essentially, incrementing it):

$$e_{\mathsf{inc}} \coloneqq \texttt{let } v = !x \texttt{ in } x \leftarrow (v + 1)$$

A reasonable specification for e_{inc} is $\{x \mapsto n\}\ e_{\mathsf{inc}}\ \{x \mapsto n + 1\}$. Here is how we can prove this specification using the proof rules in Figure 3.4:

$$\dfrac{\dfrac{}{\{x \mapsto n\}\ !x\ \{v.\, x \mapsto n * v = n\}}\ \textsc{hoare-load} \qquad \dfrac{\dfrac{\dfrac{}{\{x \mapsto v\}\ x \leftarrow (v+1)\ \{x \mapsto v + 1\}}\ \textsc{hoare-store}}{\{x \mapsto n * v = n\}\ x \leftarrow (v+1)\ \{x \mapsto v + 1 * v = n\}}\ \textsc{hoare-frame}}{\forall v.\, \{x \mapsto n * v = n\}\ x \leftarrow (v+1)\ \{x \mapsto n + 1\}}\ \textsc{hoare-csq}}{\{x \mapsto n\}\ \texttt{let } v = !x \texttt{ in } x \leftarrow (v+1)\ \{x \mapsto n + 1\}}\ \textsc{hoare-let}$$

As with inference rules, we read these so-called proof trees in a bottom-up manner. Our goal, the statement we want to prove, is at the root of the tree (the bottom). Each horizontal line represents the use of an inference rule in order to transform the current goal into one or more (hopefully simpler) goals. For example, the first rule we use in this proof is HOARE-LET, which works on let-bindings and breaks the proof into the proof of bound expression e_1 (on the left), and the proof of the let-body e_2 (on the right). Note that HOARE-LET requires the user to come up with an intermediate assertion R, which is a formula that describes the state of the program after evaluating the bound expression e_1 but before evaluating the let-body e_2. The most common way to construct R is to defer the choice until we finish the proof of the left-hand side branch, as this will give us a clue as to what R should be. In this case, since e_1 is a dereference, our only option is to use the rule HOARE-LOAD, but note that this tells us exactly what R should be.

Moving to the right branch, we now have to prove

$$\forall v.\, \{x \mapsto n * v = n\}\ x \leftarrow (v + 1)\ \{x \mapsto n + 1\}.$$

Again, looking at the program expression, our only option is to use HOARE-STORE, but the current proof goal does not quite fit. In particular, HOARE-STORE will give us the postcondition $x \mapsto v + 1$ while we need the postcondition $x \mapsto n + 1$. To infer this, we need the fact that $v = n$, which is a resource that we have in the precondition. We thus use the *frame rule* HOARE-FRAME, which lets us *frame* resources around a program fragment. This means we can carry resources that are not needed by the program fragment from its precondition to its postcondition. In our case, the program fragment is $x \leftarrow (v + 1)$, and as this does not modify v or n, we frame the resource $v = n$. This gives us a postcondition of $x \mapsto v + 1 * v = n$, which implies $x \mapsto n + 1$, so we finish the proof using HOARE-CSQ.

As you can imagine, such proof trees quickly get unweildy when reasoning about realistic programs. They also do not reflect the fact that we usually construct proofs in the same order as the program is evaluated. In this book, we represent the above proof in an inline style as follows:

1 $\{x \mapsto n\}$

```
let v = !x in
{x ↦ n * v = n}
x ← (v + 1)
{x ↦ n + 1}
```

The way to read the inline-style proof is top-down, the same way one would read a program. The proof starts with the precondition, and ends with the postcondition, and each alternate line is an intermediate assertion describing the state of the program at that point. In this example, we start off with a single heap cell $x \mapsto n$, and the first line reads the value of the heap cell into a variable v. The resulting state is thus $x \mapsto n * v = n$, containing both the heap cell and a new assertion recording the fact that $v = n$. We then write $v + 1$ to x, which results in the state $x \mapsto n + 1$.

The entire proof corresponds to a proof tree of the Hoare triple $\{x \mapsto n\}\ e_{\mathsf{inc}}\ \{x \mapsto n + 1\}$, where we combine a Hoare triple for each line using the standard proof rules for compound expressions from Figure 3.4. Intermediate assertions, such as $\{x \mapsto n * v = n\}$, are both the postcondition of the first line of code as well as the precondition of the second line of code. Note that the program expression in each line determines the proof rule that must be used to prove the triple for that line, and that each triple can be proved with this rule and standard rules like HOARE-FRAME and HOARE-CSQ. Thus, we omit the intermediate assertions for the standard rules and directly write the resulting intermediate assertion for the next line. In more complex scenarios, we use comments or multiple intermediate assertions to clarify how to get from one proof step to the next. We call the intermediate assertions in the inline-style proofs as our "proof context" at that point, because this is the set of all resources that are available at that point in the program.

For example, the simple implementation for the single-node template that we discussed before is one that uses a single heap cell for the node r. Figure 3.5 shows the definition of node in this case, which uses a default value $\bot$ to indicate that the heap cell is empty, and relates the value in the heap cell to the contents of the node if non-empty. The figure also shows a proof that the insert operation, which inserts the given value if the heap cell is empty and loops indefinitely otherwise,[3] satisfies the specification for decisive operations given in Figure 4.5.

Proving atomic triples is trickier. Recall, that an atomic triple $\langle\, P \,\rangle\ e\ \langle\, v.\ Q \,\rangle$ means there is a single physical step during the execution of e when the shared state is transformed from P to Q. Thus, while proving $\langle\, P \,\rangle\ e\ \langle\, v.\ Q \,\rangle$, we cannot treat P and Q as the pre- and postconditions of e as a whole (remember, e is potentially a complex program consisting of multiple atomic steps). It is more accurate to think of P and Q as the pre- and postcondition to e's linearization point. Unlike proofs of Hoare triples, where we are given ownership to the resources in P at the beginning of e's execution and are under an obligation to transform them to Q by the end, in proofs of atomic triples we can read or modify the resources in the precondition only during atomic steps. Furthermore, our obligation is that all atomic steps accessing P either make a modification that preserves P, except for (exactly) one step, which has to transform it into Q.

[3] All proofs in this book are concerned with *partial* correctness, i.e. if a program terminates then it satisfies the given specification. Our methodology can be extended (with additional assumptions) to also prove termination, as we discuss in the future work section of the last chapter.

```
1  node(n, C) := ∃v. r ↦ v * C = {v} \ {⊥}
2
3  {node(r, C)}
4  let insert r k =
5    {∃v. r ↦ v * C = {v} \ {⊥}}
6    let u = !r in
7    {∃v. r ↦ v * C = {v} \ {⊥} * u = v}
8    if u == ⊥ then
9      {r ↦ ⊥ * C = ∅}
10     r ← k;
11     {r ↦ k * C = ∅}
12     true
13     {node(r, {k}) * C = ∅}
14   else
15     {∃v. r ↦ v * C = {v} \ {⊥}}
16     {node(r, C)}
17     insert r k
18     {v. node(r, C') * Ψ_ω(k, C, C', v)}
19 {v. node(r, C') * Ψ_ω(k, C, C', v)}
```

Figure 3.5: Proof of insert for a single-node implementation.

$$\textsc{logatom-intro}\ \frac{\forall \Phi.\ \{\mathsf{AU}_{x.P,Q}(\Phi)\}\ e\ \{v.\ \Phi(v)\}}{\langle\, x.\ P\,\rangle\ e\ \langle\, v.\ Q\,\rangle} \qquad \textsc{logatom-atom}\ \frac{\forall x.\ \{P\}\ e\ \{v.\ Q\} \qquad e \text{ atomic}}{\langle\, x.\ P\,\rangle\ e\ \langle\, v.\ Q\,\rangle}$$

$$\textsc{au-abort}\ \frac{\langle\, x.\ P * P'\,\rangle\ e\ \langle\, v.\ P * Q'\,\rangle}{\{\mathsf{AU}_{x.P,Q}(\Phi) * P'\}\ e\ \{v.\ \mathsf{AU}_{x.P,Q}(\Phi) * Q'\}} \qquad \textsc{au-commit}\ \frac{\langle\, x.\ P * P'\,\rangle\ e\ \langle\, v.\ Q * Q'\,\rangle}{\{\mathsf{AU}_{x.P,Q}(\Phi) * P'\}\ e\ \{v.\ \Phi(v) * Q'\}}$$

$$\textsc{logatom-frame}\ \frac{\langle\, x.\ P\,\rangle\ e\ \langle\, v.\ Q\,\rangle}{\langle\, x.\ P * R\,\rangle\ e\ \langle\, v.\ Q * R\,\rangle}$$

Figure 3.6: Proof rules for establishing atomic triples.

Figure 3.6 contains the proof rules that help us execute the above proof argument. Most proofs of atomic triples start by using the rule LOGATOM-INTRO, which converts the atomic triple into a standard Hoare triple. The precondition contains an *atomic update token* $\mathsf{AU}_{x.P,Q}(\Phi)$, which records the fact that we are proving an atomic triple with precondition $x.\ P$ (recall, x is bound by a pseudo-quantifier) and postcondition Q. As we will see, this token gives us the *right* to use the resources

```
1  ⟨ lk(x) ↦ True ⟩
2    {AU_{P,Q}(Φ)} (* logatom-intro *)
3    let unlockNode x =
4      {AU_{P,Q}(Φ)}
5        ⟨ lk(x) ↦ True ⟩ (* au-commit *)
6          {lk(x) ↦ True} (* logatom-atom *)
7          lk(x) ← false
8          {lk(x) ↦ False}
9        ⟨ lk(x) ↦ False ⟩
10     {Φ}
11   {Φ}
12 ⟨ lk(x) ↦ False ⟩
```

Figure 3.7: Proof of the unlock method.

in the precondition P when executing atomic instructions, but the token also records our *obligation* to transform P to Q before execution completes. One way to use the resources in P is to use the AU-ABORT rule, which gives us access to the precondition P if the expression e is atomic: i.e. if we can prove that e atomically transforms a global (shared) precondition P and some local precondition P' into a local postcondition Q' while leaving P unchanged. This rule is useful when a program has some initial operations that modify the shared state in a way that does not change the abstract state (for instance, by locking or performing maintenance on a node). At some point, however, the program must update the shared state to the postcondition Q. LOGATOM-INTRO enforces this obligation by using an unknown, existentially quantified assertion Φ, in the postcondition of the Hoare triple. The only way to prove Φ is to use rule AU-COMMIT, which lets us exchange the atomic update token for Φ if we can prove that the program e transforms the global state from P to Q in an atomic step. As with AU-ABORT, the rule allows for some extra local resources P' and Q' in the pre- and postcondition of e.

To illustrate proofs of atomic triples, we prove that the lock and unlock methods from Figure 3.1 satisfy the following atomic specifications:

$$\langle\, b.\ \mathsf{lk}(x) \mapsto b \,\rangle\ \texttt{lockNode}\ x\ \langle\, \mathsf{lk}(x) \mapsto \mathsf{True} * b = \mathsf{False} \,\rangle$$

$$\langle\, \mathsf{lk}(x) \mapsto \mathsf{True} \,\rangle\ \texttt{unlockNode}\ x\ \langle\, \mathsf{lk}(x) \mapsto \mathsf{False} \,\rangle$$

This captures the behavior that `lockNode` atomically sets $\mathsf{lk}(x)$ to True, but only if it used to be False, and `unlockNode` sets it back to False if it used to be True.

Let us look at the simpler method, `unlockNode`, first (Figure 3.7). As mentioned above, we start by converting the atomic triple into a Hoare triple using LOGATOM-INTRO.[4] Since `unlockNode` has only a single operation, this must be the linearization point, and so we use AU-COMMIT to convert

[4]Note that for brevity we write P for the entire precondition, including any pseudo-quantifier, in all subsequent proofs.

```
1  ⟨ b. lk(x) ↦ b ⟩
2    {AU_{P,Q}(Φ)} (* logatom-intro *)
3    let lockNode x =
4      {AU_{P,Q}(Φ)}
5      ⟨ b. lk(x) ↦ b ⟩ (* au-abort or au-commit *)
6        {lk(x) ↦ b} (* logatom-atom *)
7        if CAS(lk(x), false, true) then
8          {lk(x) ↦ True ∗ b = False} (* hoare-cas-suc *)
9          ⟨ lk(x) ↦ True ∗ b = False ⟩ (* end of logatom-atom *)
10         {Φ} (* end of au-commit *)
11         ()
12       else
13         {lk(x) ↦ b} (* hoare-cas-fail *)
14         ⟨ lk(x) ↦ b ⟩ (* end of logatom-atom *)
15         {AU_{P,Q}(Φ)} (* end of au-abort *)
16         lockNode x
17         {Φ} (* Recursive call: by induction *)
18   {Φ}
19 ⟨ lk(x) ↦ True ∗ b = False ⟩
```

Figure 3.8: Proof of the lock method.

the $\mathsf{AU}_{P,Q}(\Phi)$ resource into the precondition $\mathsf{lk}(x) \mapsto \mathsf{True}$. We call this proof step of using AU-ABORT or AU-COMMIT to access resources in the precondition as *opening the precondition*. We then use LOGATOM-ATOM to perform the update on heap address $\mathsf{lk}(x)$, which gives us the state expected by the postcondition of `unlockNode`. Thus, we can successfully finish the application of the AU-COMMIT rule, which we call *committing the change*. This gives us the expected Hoare postcondition Φ (argument omitted since it is the unit), which allows us to complete the application of LOGATOM-INTRO, completing the proof.

The proof of `lockNode` is a bit more complicated (Figure 3.8). Once again, we start with LOGATOM-INTRO to convert the atomic triple to a Hoare triple. Since the first instruction in the program is a **CAS** on a location $\mathsf{lk}(x)$ in the precondition, we must open the precondition somehow. The tricky part is that we do not know whether to use AU-ABORT or AU-COMMIT because we do not know if the **CAS** will succeed until we look at the shared state in the precondition.

We thus do a proof by cases: depending on the value of b, we apply the appropriate rule (this is shown as a single proof in Figure 3.8 for brevity). In the case where b is False, the **CAS** will succeed, so we use AU-COMMIT, LOGATOM-ATOM, and HOARE-CAS-SUC to obtain the state in line 8. We can then commit the change and obtain Φ as in the previous proof. On the other hand, if b is True, then we use AU-ABORT, LOGATOM-ATOM, and HOARE-CAS-FAIL. In this case, we have not modified the shared state, so we *close* the precondition using AU-ABORT and get back the $\mathsf{AU}_{P,Q}(\Phi)$ resource. We then

deal with the recursive call to `lockNode` by using the specification that we are trying to prove,

$$\{\mathsf{AU}_{P,Q}(\Phi)\}\ \texttt{lockNode x}\ \{\Phi\},$$

to complete this branch of the proof.

CHAPTER 4

Ghost State

In this chapter, we explain the technique of using *ghost state* to construct proofs of concurrent algorithms, and describe how to do this in Iris using user-defined resources. We start by motiviating the need for ghost state using the single-node template algorithm as an example.

4.1 MOTIVATION

Recall that we want to prove that the single-node template algorithm satisfies the specification:

$$\left\langle C.\ \mathsf{CSS}(r, C) \right\rangle\ \texttt{cssOp}\ \omega\ r\ k\ \left\langle \mathsf{res}.\ \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', \mathsf{res}) \right\rangle$$

We can use the following specification for `lockNode` and `unlockNode`:

$$\left\langle b.\ \mathsf{lk}(x) \mapsto b \right\rangle\ \texttt{lockNode}\ x\ \left\langle \mathsf{lk}(x) \mapsto \mathsf{True} * b = \mathsf{False} \right\rangle$$

$$\left\langle \mathsf{lk}(x) \mapsto \mathsf{True} \right\rangle\ \texttt{unlockNode}\ x\ \left\langle \mathsf{lk}(x) \mapsto \mathsf{False} \right\rangle$$

To do this, we need a definition of the search structure predicate CSS. In §3.3.1, we proposed using an abstract predicate $\mathsf{node}(n, C_n)$ to represent the single node in the template proof, as the structure of the node (whether it consists of one heap location, or many, etc.) is implementation-specific. Building on this, a first attempt at a definition for CSS would be

$$\mathsf{CSS}_1(r, C) \coloneqq \exists b.\ \mathsf{node}(r, C) * \mathsf{lk}(r) \mapsto b,$$

which captures the single node r with contents C as well as the lock flag.

Consider Figure 4.1, a first attempt to prove `cssOp` using the above definition of CSS_1. We start by using LOGATOM-INTRO, which requires us to prove the corresponding Hoare triple instead (we use P and Q to denote the pre- and postcondition of `cssOp` for brevity). The first line of code in `cssOp` is the call to `lockNode`, whose specification tells us that it needs the resource $\mathsf{lk}(r) \mapsto \_$. As this heap cell is inside CSS_1, which is inside the precondition P, we use AU-ABORT to get access to P. Note that we do not use AU-COMMIT, since this is not the linearization point of this method. We then use LOGATOM-FRAME to frame $\mathsf{node}(r, C)$, which is not modified by `lockNode`, and reduce the proof to exactly the specification of `lockNode`. The state after `lockNode` implies P again, which allows us to complete the use of AU-ABORT, and results in the state shown on line 14.

The problem now is that the only specification we have of `decisiveOp` is

$$\left\{\mathsf{node}(r, C)\right\}\ \texttt{decisiveOp}\ \omega\ \texttt{r}\ \texttt{k}\ \left\{\mathsf{res}.\ \mathsf{node}(r, C') * \Psi_\omega(k, C, C', \mathsf{res})\right\},$$

```
1  CSS₁(r, C) ≔ ∃b. node(r, C) ∗ lk(r) ↦ b
2
3  ⟨ C. CSS₁(r, C) ⟩
4    {AU_{P,Q}(Φ)}  (* logatom-intro *)
5    let rec cssOp ω r k =
6      {AU_{P,Q}(Φ)}
7        ⟨ CSS₁(r, C) ⟩  (* au-abort *)
8        ⟨ ∃b. node(r, C) ∗ lk(r) ↦ b ⟩  (* Expand CSS₁ *)
9          ⟨ b. lk(r) ↦ b ⟩  (* logatom-frame *)
10         lockNode r;
11         ⟨ lk(r) ↦ True ⟩
12       ⟨ node(r, C) ∗ lk(r) ↦ True ⟩
13       ⟨ CSS₁(r, C) ⟩
14     {AU_{P,Q}(Φ)}
15     (* Problem: decisiveOp is not atomic! *)
16     let res = decisiveOp ω r k in
17     ...
18   {v. Φ(v)}
19 ⟨ v. CSS₁(r, C′) ∗ Ψ_ω(k, C, C′, v) ⟩
```

Figure 4.1: First attempt at a proof of the single-node template.

but our current state (line 14) does not contain the resource $\mathsf{node}(r, C)$. We cannot use AU-ABORT to open up P to get $\mathsf{node}(r, C)$ as before because `decisiveOp` is not atomic (its specification is an ordinary Hoare triple). Note that we cannot require implementations to satisfy an atomic triple for `decisiveOp`, this would defeat the purpose of the template, which locks the node before calling `decisiveOp` in order to allow a sequential implementation whose correctness proof only involves sequential reasoning.

We solve this issue (Figure 4.2) by adapting the definition of CSS to reflect the protocol followed by all threads in this algorithm: that threads modify a node only when they have locked it. Consider the alternative definition:

$$\mathsf{CSS}_2(r, C) \coloneqq \exists b.\ \mathsf{lk}(r) \mapsto b * (b\ ?\ \mathsf{True} : \mathsf{node}(r, C))$$

Here, the second term is $\mathsf{node}(r, C)$ if b is false (the node is unlocked), indicating that unlocked nodes belong to the shared state and are available for anyone to acquire. But if b is true (the node is locked), then the shared state contains only True, which means that the thread that locked the node can take possession of the $\mathsf{node}(r, C)$ resource into its local state. Concretely, this means that the proof context will contain $\mathsf{CSS}_2(r, C) * \mathsf{node}(r, C)$ when we finish the proof of the Hoare triple in the premise of rule AU-ABORT (line 13). Note that we have here instantiated the rule with $P = \mathsf{CSS}_2(r, C)$ and $Q' = \mathsf{node}(r, C)$. Since we only need to "give back" ownership of P in AU-ABORT, our context at line 14 will be $\mathsf{AU}_{P,Q}(\Phi) * \mathsf{node}(r, C)$. We are now back to proving a Hoare

```
1  CSS₂(r, C) := ∃b. lk(r) ↦ b * (b ? True : node(r, C))
2
3  ⟨ C. CSS₂(r, C) ⟩
4    {AU_{P,Q}(Φ)} (* logatom-intro *)
5    let rec cssOp ω r k =
6      {AU_{P,Q}(Φ)}
7        ⟨ CSS₂(r, C) ⟩ (* au-abort *)
8        ⟨ ∃b. lk(r) ↦ b * (b ? True : node(r, C)) ⟩ (* Expand CSS₂ *)
9          ⟨ b. lk(r) ↦ b ⟩ (* logatom-frame *)
10         lockNode r;
11         ⟨ lk(r) ↦ True * b = False ⟩
12       ⟨ lk(r) ↦ True * node(r, C) ⟩
13       ⟨ CSS₂(r, C) * node(r, C) ⟩
14     {AU_{P,Q}(Φ) * node(r, C)}
15       {node(r, C)} (* hoare-frame *)
16       let res = decisiveOp ω r k in
17       {node(r, C') * Ψ_ω(k, C, C', res)}
18     {AU_{P,Q}(Φ) * node(r, C') * Ψ_ω(k, C, C', res)}
19       ⟨ CSS₂(r, C'') * node(r, C') * Ψ_ω(k, C, C', res) ⟩ (* au-commit *)
20         ⟨ ∃b. lk(r) ↦ b * (b ? True : node(r, C)) * node(r, C') * Ψ_ω(k, C, C', res) ⟩ (* Expand CSS₂ *)
21         ⟨ lk(r) ↦ True * node(r, C') * Ψ_ω(k, C, C', res) ⟩ (* Property of node *)
22           ⟨ lk(r) ↦ True ⟩ (* logatom-frame *)
23           unlockNode r;
24           ⟨ lk(r) ↦ False ⟩
25         ⟨ lk(r) ↦ False * node(r, C') * Ψ_ω(k, C, C', res) ⟩
26         ⟨ CSS₂(r, C') * Ψ_ω(k, C, C', res) ⟩
27         (* Problem: C ≠ C'' *)
28       ⟨ CSS₂(r, C') * Ψ_ω(k, C'', C', res) ⟩
29     {Φ(res)}
30     ...
31   {v. Φ(v)}
32 ⟨ v. CSS₂(r, C') * Ψ_ω(k, C, C', v) ⟩
```

Figure 4.2: Second attempt at a proof of the single-node template.

triple and can apply the Hoare triple specification of `decisiveOp` on $\mathsf{node}(r, C)$. We think of any resources we have in an intermediate state with curly braces, such as $\{\mathsf{node}(r, C)\}$, as being in the local state of the current thread, which lets us perform non-atomic operations on $\mathsf{node}(r, C)$ without risk of interference (i.e. without the risk that some other thread will read or modify that local state).

Figure 4.2 presents a second proof attempt that shows this locking mechanism in action. As described above, when the application of AU-ABORT around the call to `lockNode` concludes, we have

$\mathsf{node}(r, C)$ in the local state, and thus we can use the specification of `decisiveOp` (after using HOARE-FRAME to frame the extra resources $\mathsf{AU}_{P,Q}(\Phi)$).

We now turn our attention to the call to `unlockNode`. Note that `unlockNode` modifies the lock location of r, whose heap cell is still part of the shared state (as it is contained in the definition of CSS_2), so we need to open the precondition again. But this step in the algorithm is also its linearization point, because when the thread unlocks the node that it has (potentially) modified, any changes it has made become visible to other threads.[1] We thus use the rule AU-COMMIT to open the precondition, which brings us to our second problem: recall that the precondition is $\langle\, C.\ \mathsf{CSS}_2(r, C) \,\rangle$, where C is (pseudo) quantified. This means that every time we open the precondition, we get a potentially different set of contents C. Since we already have a variable called C in our context, when opening the precondition we get $\mathsf{CSS}_2(r, C'')$ using a fresh variable C'' on line 19. The reason is that $\mathsf{CSS}_2(r, C)$ is in the *shared* state, and theoretically other threads can modify the contents of the search structure between the last time we opened the precondition (at the call to `lockNode`) and now. Although this cannot happen in our single-node example, where the current thread has locked the only node in the structure, the proof rules are generic rules that must apply to all algorithms, including search structures consisting of multiple nodes.

Unfortunately, AU-COMMIT needs us to show that `lockNode` results in the postcondition of `cssOp` where the predicate $\Psi_\omega(k, C'', C', \mathsf{res})$ uses C'', the same variable used in the precondition (line 28). However, we have $\Psi_\omega(k, C, C', \mathsf{res})$, which we got from the postcondition of `decisiveOp`. The last time we looked, during the call to `lockNode`, the contents of the shared state was C. Thus, there is no way we can prove $\Psi_\omega(k, C'', C', \mathsf{res})$.

The real problem is in the definition of CSS_2:

$$\mathsf{CSS}_2(r, C) \coloneqq \exists b.\ \mathsf{lk}(r) \mapsto b * (b\ ?\ \mathsf{True} : \mathsf{node}(r, C))$$

Note that when the single node it contains is locked, this formula becomes

$$\mathsf{lk}(r) \mapsto \mathsf{True} * \mathsf{True},$$

which says nothing about the contents C. This is why when we access the shared state for the second time in line 19, we know nothing about the contents C''. To fix this, we need to modify the definition of CSS to reflect the fact that since it has only one node, if that node is locked, then the contents cannot be changed. On the other hand, when the node is unlocked, and put back into CSS, then the contents can be updated. To do this, we will need to use a new concept, *ghost state*.

4.2 GHOST STATES AND RESOURCE ALGEBRAS

Ghost state, originally called auxiliary variables [Owicki and Gries, 1976], is a formal technique where the prover adds state (variables or resources) to a program that capture knowledge about the

[1] In fact, the linearization point will be at the point of unlocking for all the lock-based single-copy search structures that we study in this book. In chapters 8 and 9 we will study templates where this is not the case.

history of a computation not present in the state of the original program in order to verify it. As long as the added ghost state, and the ghost commands that modify it, have no effect on the run-time behavior of the program, then a so-called *erasure* theorem states that a proof of the augmented program can be transformed into a proof of the program with all ghost state removed (i.e., the original program). The ghost state concept has shown itself to be an invaluable tool in the verifier's toolbox, and has been used to encode many common reasoning techniques including permissions, tokens, capabilities, and protocols.

Iris expresses ownership of ghost state by the proposition $\boxed{a}^{\gamma}$ which asserts ownership of a piece a of the ghost location γ. It is the ghost analogue of the points-to predicate $x \mapsto v$ that asserts that the (real) location x contains value v, except that $\boxed{a}^{\gamma}$ asserts only that γ contains a value *one of whose parts* is a. This means ghost state can be split and combined according to the rules of the *resource algebra* (RA), or the algebraic structure, from which the values (like a) are drawn. We define RAs formally below, but for now, think of an RA as a set M, with a *validity* predicate $\overline{\mathcal{V}}(-)$, and a binary operation $(\cdot)\colon M \times M \to M$. The two important mechanisms for using ghost state are: (1) a ghost location can be split and combined using the rule: $\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$; and (2) at any point, if a thread owns a resource $\boxed{c}^{\gamma}$ then the value c is valid ($\overline{\mathcal{V}}(c)$).

What does it mean to split a ghost location, and why is this needed? Coming back to our single-node template example, we need a way to keep track of the contents of the structure, both in the shared state and in the local state of the thread that locks the node in such a way that the two "views" of the contents are the same. This can be achieved by using the concept of fractional permissions: we associate a fraction with a ghost location, so that the ghost location can be split and shared among many threads to permit shared reads, but threads can only write to the ghost location if they own the full ghost location. We achieve this by using a *fractional* RA (defined formally below), that consists of elements of the form (q, C) for any fraction $q \in (0, 1]$ and any set of keys C, as well as a special $\lightning$ element. Composition in this RA is defined as $(q_1, C_1) \cdot (q_2, C_2) = (q_1 + q_2, C_1)$ if $C_1 = C_2$ and $q_1 + q_2 \leq 1$ and $\lightning$ otherwise. Validity is defined as $\overline{\mathcal{V}}(a) = (a \neq \lightning)$, making $\lightning$ the only invalid element. The upshot of these definitions is that in our example, we can extend the definition of $\mathsf{CSS}(r, C)$ to

$$\mathsf{CSS}(r, C) \coloneqq \exists b.\, \boxed{½\, C}^{\gamma} * \mathsf{lk}(r) \mapsto b * \left(b \;?\; \mathsf{True} : \mathsf{node}(r, C) * \boxed{½\, C}^{\gamma}\right),$$

where we write $\boxed{½\, C}^{\gamma}$ for $\boxed{(½, C)}^{\gamma}$ for brevity. When a thread locks r, we give it both $\mathsf{node}(r, C)$ and one of the halves $\boxed{½\, C}^{\gamma}$. This way, when the thread looks at the shared state later, say when unlocking r, it will have $\boxed{½\, C''}^{\gamma} * \boxed{½\, C}^{\gamma}$ and since all resources have to be valid this means $C = C''$.

Formally, a resource algebra is a generalization of the partial commutative monoid (PCM) algebra commonly used by separation logics. The definition of an RA is given in Figure 4.3, where

A *resource algebra* is a tuple $(M, \overline{\mathcal{V}}\colon M \to \mathit{Prop}, |-|\colon M \to M^?, (\cdot)\colon M \times M \to M)$ satisfying:

$$\begin{aligned}
&\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) && \text{(RA-ASSOC)}\\
&\forall a, b.\ a \cdot b = b \cdot a && \text{(RA-COMM)}\\
&\forall a.\ |a| \in M \Rightarrow |a| \cdot a = a && \text{(RA-CORE-ID)}\\
&\forall a.\ |a| \in M \Rightarrow ||a|| = |a| && \text{(RA-CORE-IDEM)}\\
&\forall a, b.\ |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| && \text{(RA-CORE-MONO)}\\
&\forall a, b.\ \overline{\mathcal{V}}(a \cdot b) \Rightarrow \overline{\mathcal{V}}(a) && \text{(RA-VALID-OP)}
\end{aligned}$$

where $M^? := M \uplus \{\bot\}$ $\qquad a^? \cdot \bot := \bot \cdot a^? := a^?$

$a \preccurlyeq b := \exists c \in M.\ b = a \cdot c$

Figure 4.3: The definition of a resource algebra (RA).

Prop is the type of propositions of the meta-logic (e.g. Coq).[2] The composition operator $\cdot$ must be associative and commutative. The partial function $|-|$ assigns to an element a a *core* $|a|$ (which can be thought of as a's own unit, see RA-CORE-ID). For technical reasons, the core function must be idempotent and monotonic. As mentioned previously, RAs use a validity predicate $\overline{\mathcal{V}}$ to identify valid elements of the domain. [3] The rule RA-VALID-OP disallows taking an invalid element and composing it with another element to make it valid; since Iris maintains an invariant that the composition of all values in a ghost location is valid, this rule implies that any sub-resource in that location is also valid.

Example 4.1 Given a set S, we define the fractional RA over values of S as:

$$M := (\mathbb{Q} \cap (0, 1], S) \mid \lightning \qquad \overline{\mathcal{V}}(a) := a \neq \lightning \qquad |(q, s)| := |\lightning| := \lightning$$

$$(q_1, s_1) \cdot (q_2, s_2) := \begin{cases} (q_1 + q_2, s_1) & \text{if } q_1 + q_2 \leq 1 \land s_1 = s_2 \\ \lightning & \text{otherwise} \end{cases} \qquad \lightning \cdot \_ := \_ \cdot \lightning := \lightning$$

Ghost state by itself is not very useful unless it can be updated. However, unlike physical state, which can be modified at any point to any value, ghost state updates are restricted since Iris

[2] Iris actually uses *cameras* as the structure underlying resources, but as we do not use higher-order resources (i.e. state which can embed propositions) in this book we restrict our attention to RAs, a stronger, but simpler, structure.

[3] Readers familiar with separation algebras will notice that the composition operator is not partial; cases where composition used to be undefined can be encoded by sending them to an invalid element.

maintains the invariant that the composition of all the pieces of ghost state at a particular location is valid (as given by $\overline{\mathcal{V}}$). Iris allows only *frame-preserving updates* $a \rightsquigarrow b$, defined below.

Definition 4.2 A *frame-preserving update* is a relation between an element $a \in M$ and a set $B \subseteq M$, written $a \rightsquigarrow B$, such that

$$\forall a_f^? \in M^?.\ \overline{\mathcal{V}}(a \cdot a_f^?) \Rightarrow \exists b \in B.\ \overline{\mathcal{V}}(b \cdot a_f^?).$$

We write $a \rightsquigarrow b$ if $a \rightsquigarrow \{b\}$.

Intuitively, $a \rightsquigarrow b$ says that every *frame* a_f that is compatible with a should also be compatible with b. Thus, changing a thread's fragment of the ghost state from a to some b will not invalidate assumptions about a_f made by any other thread. The fractional RA has the following frame-preserving update:

$$\text{FRAC-UPD}\quad (1, s) \rightsquigarrow (1, s')$$

Note that the element $(1, s)$ has no frame (no other element can compose with it), thus the frame-preserving update condition holds trivially.

This allows us to change the value stored at a ghost location as long as we own all the pieces of that location. Correspondingly, we also note that there are no frame-preserving updates from (q, s) when $q < 1$, which means no thread can change the value unless that thread holds all the fragments.

4.3 PROOF OF THE SINGLE-NODE TEMPLATE

We finally have all the pieces needed to prove the single-node template (Figure 4.4). Since all the proofs of atomic triples in this book follow the same structure of using LOGATOM-INTRO to turn it into a Hoare triple proof and then AU-ABORT and AU-COMMIT for each operation in the program, we omit those details from the proof sketch. We also omit the $\mathsf{AU}_{P,Q}(\Phi)$ since it appears on every line, and instead just assume we have the resources in the precondition when we go from a Hoare triple to an atomic triple, like on line 7. We also use an additional predicate $\mathsf{N}(n, C)$ for the combination of $\mathsf{node}(n, C)$ and the ghost state associated with that node.

The call to `lockNode` is handled as before, except that this time we take out $\mathsf{N}(r, C)$ from the precondition after locking r. The call to `decisiveOp` is also handled as before, and note that it modifies only the contents in $\mathsf{node}(r, C')$, the ghost state $\boxed{\tfrac{1}{2}\ C}^{\gamma}$ continues to record the contents of the shared state from the time when the node was locked. When we open the precondition around the call to `unlockNode`, we get the state shown on line 14. First, we use the property of the abstract predicate `node` that $\mathsf{node}(n, C_n) * \mathsf{node}(n, C'_n) \mathrel{-\!\!*} \mathsf{False}$ to infer that b must be True (line 15). Second, we use the fractional resource algebra properties on the ghost state the thread owns at this point, $\boxed{\tfrac{1}{2}\ C''}^{\gamma} * \boxed{\tfrac{1}{2}\ C}^{\gamma}$, to infer that $C = C''$ (line 16). Third, we perform a frame-preserving update using FRAC-UPD to convert the ghost state to $\boxed{\tfrac{1}{2}\ C'}^{\gamma} * \boxed{\tfrac{1}{2}\ C'}^{\gamma}$, reflecting the updated contents

```
1  N(n, C) := [½ C]^γ * node(n, C)
2  CSS(r, C) := [½ C]^γ * ∃b. lk(n) ↦ b * (b ? True : N(r, C))
3
4  ⟨ C. CSS(r, C) ⟩
5  let rec cssOp ω r k =
6    {True}
7      ⟨ [½ C]^γ * ∃b. lk(n) ↦ b * (b ? True : N(r, C)) ⟩
8      lockNode r;
9      ⟨ [½ C]^γ * lk(n) ↦ True * N(r, C) ⟩
10   {N(r, C)}
11   { node(r, C) * [½ C]^γ }
12   let res = decisiveOp ω r k in
13   { node(r, C′) * [½ C]^γ * Ψω(k, C, C′, res) }
14     ⟨ [½ C″]^γ * ∃b. lk(n) ↦ b * (b ? True : N(r, C″)) * N(r, C′) * Ψω(k, C, C′, res) ⟩
15     ⟨ [½ C″]^γ * lk(n) ↦ True * node(r, C′) * [½ C]^γ * Ψω(k, C, C′, res) ⟩
16     ⟨ [½ C]^γ * lk(n) ↦ True * node(r, C′) * [½ C]^γ * Ψω(k, C, C′, res) ⟩
17     ⟨ [½ C′]^γ * lk(n) ↦ True * node(r, C′) * [½ C′]^γ * Ψω(k, C, C′, res) ⟩
18     unlockNode r;
19     ⟨ [½ C′]^γ * lk(n) ↦ False * node(r, C′) * [½ C′]^γ * Ψω(k, C, C′, res) ⟩
20     ⟨ CSS(r, C′) * Ψω(k, C, C′, res) ⟩
21   {True}
22   res
23 ⟨ v. CSS(r, C′) * Ψω(k, C, C′, v) ⟩
```

Figure 4.4: Proof of the single-node template algorithm.

$$\{\mathsf{node}(r, C)\}\ \texttt{decisiveOp}\ \omega\ \texttt{r}\ \texttt{k}\ \{v.\ \mathsf{node}(r, C') * \Psi_\omega(k, C, C', v)\}$$

$$\mathsf{node}(n, C_n) * \mathsf{node}(n, C'_n) \mathrel{-\!\!*} \mathsf{False}$$

Figure 4.5: The assumptions made by the single-node template on implementations.

of the search structure. We can then use the atomic specification of `unlockNode` to get the state shown on line 19. This can be rewritten to obtain the postcondition, completing the proof.

The assumptions made by this template proof on implementation-specific helper functions and abstract predicates are listed in Figure 4.5. These functions and predicates are defined by imple-

```
1 let rec cssOp ω n1 n2 k =
2   let n = findNode n1 n2 k in
3   lockNode n;
4   let res = decisiveOp ω n k in
5   unlockNode n;
6   res
```

Figure 4.6: A template algorithm for a two-node search structure.

mentations, which must satisfy the given assumptions. For example, we saw in §3.3.3 a proof that the simple implementation for the single-node template that uses a single heap cell for the node r satisfies the decisive operation specification.

Next, we will see how to extend these ideas to a structure consisting of more than one node.

4.4 TWO-NODE TEMPLATE AND KEYSETS

We now look at a search structure that contains two nodes, n_1 and n_2, whose template algorithm is listed in Figure 4.6. Since there are two nodes, the first step in this algorithm is to find the node in which to search for, insert, or delete the given key. This is done via a new helper function `findNode`. Once the appropriate node n is found, the algorithm proceeds similarly to the single-node template: it locks n, calls `decisiveOp` on it, and then unlocks it.

Implementations of this template choose not only how to store the keys in a node (e.g. as an array of keys or a list of keys) but also how to divide keys between nodes. For instance, one possible implementation would be to send the odd keys to n_1 and the even keys to n_2. We represent this choice in the template proof via an abstract function[4] $\mathsf{ks}(n)$ that maps a node n to a set of keys we call the *keyset*. Intuitively, we expect the implementation to define the keyset of a node n as the set of keys $\mathsf{ks}(n)$ that, if present in the structure, must be in n. In the above example, $\mathsf{ks}(n_1)$ is the set of odd numbers, and $\mathsf{ks}(n_2)$ is the set of even numbers. The proof of the template can use this keyset function to specify the behavior it expects from the `findNode` helper function:

$$\{\mathsf{True}\}\ \texttt{findNode}\ n_1\ n_2\ k\ \{n.\ \mathsf{inFP}(n_1, n_2, n) * k \in \mathsf{ks}(n)\}$$

Here, $\mathsf{inFP}(n_1, n_2, n) \coloneqq (n = n_1 \vee n = n_2)$ is a predicate that captures the fact that n is in the *footprint* of the data structure, i.e. that it is one of the nodes in the data structure.[5] We continue to use the simple spin-lock implementation of `lockNode` and `unlockNode`, which, as shown in Chapter 3, satisfy the following specifications:

$$\langle b.\ \mathsf{lk}(x) \mapsto b \rangle\ \texttt{lockNode}\ x\ \langle \mathsf{lk}(x) \mapsto \mathsf{True} * b = \mathsf{False} \rangle$$

[4] Abstract functions are like abstract predicates in that the template proof is done without knowing their definition; instead, the proof relies on certain assumptions about them.

[5] While this is a trivial definition for the two-node template, we will use the same predicate to simplify the more complex proofs in later chapters.

$$\left\langle \mathsf{lk}(x) \mapsto \mathsf{True} \right\rangle \ \texttt{unlockNode}\ x \ \left\langle \mathsf{lk}(x) \mapsto \mathsf{False} \right\rangle$$

Note that a thread can call `lockNode` or `unlockNode` only on a node x for which it owns the heap cell $\mathsf{lk}(x)$ – this is where the $\mathsf{inFP}(n_1, n_2, n)$ predicate will be used.

The challenge is in providing a suitable specification for `decisiveOp`. At the point when `decisiveOp` is called, only one of the two nodes in the structure is locked by the current thread, and hence any specification for `decisiveOp` can speak only about the node n. A natural first-attempt would be:

$$\left\{\mathsf{node}(n, C_n)\right\}\ \texttt{decisiveOp}\ \omega\ n\ k\ \left\{\mathsf{res}.\ \mathsf{node}(n, C_n') * \Psi_\omega(k, C_n, C_n', \mathsf{res})\right\},$$

This spec says `decisiveOp` converts $\mathsf{node}(n, C_n)$ (node n with contents C_n) into $\mathsf{node}(n, C_n')$ (the node n with updated contents C_n') such that the search structure specification predicate $\Psi_\omega(k, C_n, C_n', \mathsf{res})$ holds. However, the trouble is that the postcondition of `cssOp` requires us to show that the contents of the entire search structure are modified from some C to C' such that $\Psi_\omega(k, C, C', \mathsf{res})$ holds. To complete the proof, we need to show that $\Psi_\omega(k, C_n, C_n', \mathsf{res}) \Rightarrow \Psi_\omega(k, C, C', \mathsf{res})$.

This is not true of arbitrary sets $C_n \subseteq C$ and $C_n' \subseteq C'$. Consider the case where node n_1 has contents $\{1, 3, 8\}$, and n_2 has contents $\{2, 4, 8\}$ and `decisiveOp` removes key 8 from n_1. Here $C_n = \{1, 3, 8\}$ and $C_n' = \{1, 3\}$, but $C = C' = \{1, 2, 3, 4, 8\}$.

So, we need more constraints. Our example implementation assigned each node a distinct set of keys (n_1 got the odd keys and n_2 got even keys). The missing piece of the proof is the property that the keysets of any two nodes are *disjoint*. If we have a data structure where all keysets are disjoint and the contents of each node n are a subset of the keyset of n, then we can show that it is sufficient for `decisiveOp` to ensure that Ψ_ω holds on the node n such that $k \in \mathsf{ks}(n)$. We next show how to encode this argument in separation logic using a novel Iris resource algebra.

4.5 DISJOINT KEYSETS AND THE KEYSET RA

In order to define a resource algebra (RA) for disjoint keysets, we first describe some standard RA constructions that are useful in defining more complex RAs [Iris Team, 2020, Jung et al., 2018].

Definition 4.3 Given a set S, the *exlusive* RA $\mathrm{Ex}(S)$ is defined as:

$$\mathrm{Ex}(S) := \mathsf{ex}(S) \mid \lightning \qquad \overline{\mathcal{V}}(a) := (a \neq \lightning) \qquad |\mathsf{ex}(x)| := \bot \qquad |\lightning| := \lightning \qquad \_ \cdot \_ := \lightning$$

This RA is called the exclusive RA because it is defined such that at most one $x \in S$ can be owned. This is why: Iris is designed in such a way that the composition of all resources owned at any point is valid. Since the composition of any two elements of the exclusive RA is $\lightning$, which is

invalid, this means at most one $x \in S$ can be owned by any thread at a given point in time. We thus obtain the following frame-preserving update:

$$\frac{}{\mathsf{ex}(x) \rightsquigarrow \mathsf{ex}(y)}\ \textsc{ex-update}$$

This captures the intuition that since the resource is exclusive, if one thread owns it, then no other thread has access to it; hence the thread can change it to anything it likes.

The second RA construction we use is the *authoritative* RA $\textsc{Auth}(M)$, constructed from any other RA M. This is used to model situations where one thread owns an authoritative element a of M, and others potentially own fragments $b \preccurlyeq a$ of a.

Definition 4.4 Given an RA M that has a unit ε (and hence, a total core), the *authoritative* RA $\textsc{Auth}(M)$ is defined as:

$$\textsc{Auth}(M) \coloneqq (\textsc{Ex}(M) \mid \bot) \times M$$

$$\overline{\mathcal{V}}((x, b)) \coloneqq (x = \bot \land \overline{\mathcal{V}}(b)) \lor (\exists a.\, x = \mathsf{ex}(a) \land b \preccurlyeq a \land \overline{\mathcal{V}}(a))$$

$$(x_1, b_1) \cdot (x_2, b_2) \coloneqq (x_1 \cdot x_2, b_2 \cdot b_2) \qquad |(x, b)| \coloneqq (\bot, |b|)$$

Let $a, b \in M$. When using the $\textsc{Auth}(M)$ RA, we write $\bullet\, a$ for full ownership $(\mathsf{ex}(a), \varepsilon)$ and $\circ\, b$ for fragmental ownership $(\bot, b)$ and $\bullet\, a, \circ\, b$ for combined ownership $(\mathsf{ex}(a), b)$. A simple property of authoritative RAs that we will use is:

$$\frac{}{(\circ\, a) \cdot (\circ\, b) = \circ\, (a \cdot b)}\ \textsc{auth-frag-op}$$

This rule says that we can split or combine a fragment according to the composition operator of the underlying RA.

We can now define the RA that we use to keep track of the keyset and contents of each node.

Definition 4.5 Given a key space KS, the *keyset* RA is defined as:

$$\textsc{Keyset} \coloneqq (\mathsf{KS} \times \mathsf{KS}) \mid \lightning \mid \bot \qquad \overline{\mathcal{V}}((K, C)) \coloneqq (C \subseteq K) \qquad \overline{\mathcal{V}}(\_) \coloneqq \mathsf{False}$$

$$(K_1, C_1) \cdot (K_2, C_2) \coloneqq \begin{cases} (K_1 \cup K_2, C_1 \cup C_2) & \text{if } C_1 \subseteq K_1 \land C_2 \subseteq K_2 \land K_1 \cap K_2 = \emptyset \\ \lightning & \text{otherwise} \end{cases}$$

$$\lightning \cdot \_ \coloneqq \_ \cdot \lightning \coloneqq \lightning \qquad \bot \cdot a \coloneqq a \cdot \bot \coloneqq a \qquad |a| \coloneqq \bot$$

This is an RA where elements are pairs of sets of keys, where the first set represents the keyset and the second represents the contents of a node (or, more generally, a set of nodes). We also have two special elements, $\lightning$ representing invalid compositions and a unit $\bot$, so that we can use an authoritative version of this RA below. The validity predicate checks if the contents are a subset of the keyset, and composition is only defined between valid elements whose keysets are disjoint.

In our proofs, we will be using Auth(Keyset), the authoritative keyset RA. Recall that for the single-node template proof (§4.3), we needed to use the fractional RA so that the shared state could keep track of the global contents even when the single node was locked. Similarly, we need two copies of the ghost state that keeps track of the contents, one copy that always stays in the shared state and is fixed to equal the parameter C in $\mathsf{CSS}(n_1, n_2, C)$, and one copy that is split among nodes and handed out to threads who lock a node. Performing this reasoning with many fractional locations gets messy and tedious, so we instead use the authoritative RA, which was built for such situations.

Using Auth(Keyset), we add the formula $\boxed{\bullet(\mathsf{KS}, C)}^{\gamma}$ to the definition of CSS to represent the abstract state of the search structure as one whose keyset is the entire key space KS and contains the keys C. Similarly, we represent the local abstract state of a node n by the formula $\boxed{\circ(K_n, C_n)}^{\gamma}$, where K_n and C_n are the keyset and contents, respectively, of n. By the definition of the authoritative RA, the assertion

$$\boxed{\bullet(\mathsf{KS}, C)}^{\gamma} * \mathop{\Large *}_{n \in N} \boxed{\circ(K_n, C_n)}^{\gamma}$$

expresses that the sets K_n for each $n \in N$ are disjoint and their union is included in KS . Moreover, $C_n \subseteq K_n$ and similarly the C_n sets are disjoint and are included in C. If we can associate each C_n and K_n to the contents and keyset, respectively, of n, then an assertion like the one above gives us the desired disjoint decomposition of the abstract state into local states.

The Auth(Keyset) RA has frame-preserving updates such as the following, which we will use to update the ghost state when we insert or delete a key k:

KS-INS
$$\frac{\overline{\mathcal{V}}((K, C)) \qquad \overline{\mathcal{V}}((K_n, C_n)) \qquad k \in K_n}{\bullet(K, C), \circ(K_n, C_n) \rightsquigarrow \bullet(K, C \cup \{k\}), \circ(K_n, C_n \cup \{k\})}$$

KS-DEL
$$\frac{\overline{\mathcal{V}}((K, C)) \qquad \overline{\mathcal{V}}((K_n, C_n)) \qquad k \in K_n}{\bullet(K, C), \circ(K_n, C_n) \rightsquigarrow \bullet(K, C \setminus \{k\}), \circ(K_n, C_n \setminus \{k\})}$$

For example, KS-DEL says that if $\boxed{\bullet(K, C)}^{\gamma}$ and $\boxed{\circ(K_n, C_n)}^{\gamma}$ are valid resources such that $k \in K_n$ then we can update the fragment to $(K_n, C_n \setminus \{k\})$ (for instance when we remove k from the contents of a node n) and the authoritative resource to $(K, C \setminus \{k\})$ (meaning k is also removed from the global contents). Combining this with KS-INS for insertions, we get the following lemma:

KS-UPD
$$\boxed{\bullet(K, C)}^{\gamma} * \boxed{\circ(K_n, C_n)}^{\gamma} * k \in K_n * \Psi_\omega(k, C_n, C'_n, \mathsf{res}) \Rrightarrow \exists C'. \boxed{\bullet(K, C')}^{\gamma} * \boxed{\circ(K_n, C'_n)}^{\gamma} * \Psi_\omega(k, C, C', \mathsf{res})$$

$\mathsf{inFP}(n_1, n_2, n) \coloneqq (n = n_1 \lor n = n_2)$

$\mathsf{N}(n) \coloneqq \exists C_n.\ \mathsf{node}(n, C_n) * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma}$

$\mathsf{CSS}(n_1, n_2, C) \coloneqq \boxed{\bullet(\mathsf{KS}, C)}^{\gamma} * (\exists b_1.\ \mathsf{lk}(n_1) \mapsto b_1 * (b_1\ ?\ \mathsf{True} : \mathsf{N}(n_1)))$
$\qquad * (\exists b_2.\ \mathsf{lk}(n_2) \mapsto b_2 * (b_2\ ?\ \mathsf{True} : \mathsf{N}(n_2)))$

$\langle C.\ \mathsf{CSS}(n_1, n_2, C) \rangle$
```
let rec cssOp ω r k =
```
$\{\mathsf{True}\}$
```
  let n = findNode n1 n2 k in
```
$\{\mathsf{inFP}(n_1, n_2, n) * k \in \mathsf{ks}(n)\}$
```
  lockNode n;
```
$\{\mathsf{N}(n)\}$

$\{\mathsf{node}(n, C_n) * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma}\}$
```
  let res = decisiveOp ω n k in
```
$\{\mathsf{node}(n, C'_n) * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma} * \Psi_\omega(k, C_n, C'_n, \mathsf{res})\}$

$\langle \mathsf{node}(n, C'_n) * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma} * \Psi_\omega(k, C_n, C'_n, \mathsf{res}) * \boxed{\bullet(\mathsf{KS}, C)}^{\gamma} * \cdots \rangle$

$\langle \mathsf{node}(n, C'_n) * \boxed{\circ(\mathsf{ks}(n), C'_n)}^{\gamma} * \Psi_\omega(k, C, C', \mathsf{res}) * \boxed{\bullet(\mathsf{KS}, C')}^{\gamma} * \cdots \rangle$ (* By KS-UPD *)
```
  unlockNode n;
```
$\langle \mathsf{CSS}(n_1, n_2, C') * \Psi_\omega(k, C, C', \mathsf{res}) \rangle$

$\{\mathsf{True}\}$
```
  res
```
$\langle v.\ \mathsf{CSS}(n_1, n_2, C') * \Psi_\omega(k, C, C', v) \rangle$

Figure 4.7: Proof of the two-node template algorithm.

$$\{\mathsf{True}\}\ \texttt{findNode}\ n_1\ n_2\ k\ \{n.\ \mathsf{inFP}(n_1, n_2, n) * k \in \mathsf{ks}(n)\}$$

$$\{\mathsf{node}(n, C_n)\}\ \texttt{decisiveOp}\ \omega\ n\ k\ \{\mathsf{res}.\ \mathsf{node}(n, C'_n) * \Psi_\omega(k, C_n, C'_n, \mathsf{res})\}$$

$$\mathsf{node}(n, C_n) * \mathsf{node}(n, C'_n) \mathrel{-\!\!*} \mathsf{False}$$

Figure 4.8: The assumptions made by the two-node template on implementations.

This lemma is expressed in terms of Iris' basic update modality $\dot{\Rightarrow}$. The intuitive meaning of $P \dot{\Rightarrow} Q$ is that if we have the resource P then we can do a ghost state update and get Q.

We can now prove the two-node template (Figure 4.7). The definition of CSS has been extended to account for two nodes and a locking mechanism for each. It also contains the authoritative version of the keyset and global contents: $\boxed{\bullet(\mathsf{KS}, C)}^{\gamma}$. Each node is represented by the node pred-

icate $\mathsf{N}(n)$, which contains the abstract predicate $\mathsf{node}(n, C_n)$ that is implementation-specific as well as the fragment containing n's keyset and contents $\boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma}$.

The call to `findNode` is handled as explained previously, using the specification given in Figure 4.8. To prove the precondition of `lockNode`, we now need to show that we own $\mathsf{lk}(n)$, which we can do using the prediate $\mathsf{inFP}(n_1, n_2, n)$ that we obtained from `findNode`. After `lockNode`, we can move the node $\mathsf{N}(n)$ from the shared state into our local state as before. We then use the specification of `decisiveOp` to get a modified node predicate $\mathsf{node}(n, C'_n)$ and $\Psi_\omega(k, C_n, C'_n, \mathsf{res})$.

As with the single-node template, the linearization point is at the call to `unlockNode`. We use the rule AU-COMMIT to open the precondition and get access to the shared state, obtaining the resources shown in the intermediate assertion on line 16. We then use KS-UPD to udpate both the node's fragment of the keyset RA as well as the authoritative element to the new contents, and obtain the resource $\Psi_\omega(k, C, C', \mathsf{res})$. This step corresponds to the reasoning that since the decisive operation was performed on a node n such that $k \in \mathsf{ks}(n)$, the global contents also change appropriately. We can then apply `unlockNode`'s specification to change the lock location of n, and return $\mathsf{N}(n)$ to the shared state, obtaining the postcondition $\mathsf{CSS}(n_1, n_2, C') * \Psi_\omega(k, C, C', \mathsf{res})$.

CHAPTER 5

The Flow Framework

So far, we have seen how to use ghost state in order to verify template algorithms for one and two-node structures. Most useful real-world structures, like the B-link tree (Chapter 2), have a dynamic and unbounded number of nodes. Extending our proofs to structures with unbounded nodes presents a challenge, because we want to prove that a thread preserves global invariants while doing operations on a few nodes. We formulate such proofs using the *flow framework*. This chapter defines and motivates the flow framework.

5.1 MOTIVATION

To see why the two-node template proof does not extend to the unbounded-node setting, recall that we represented the shared state using the CSS predicate, defined as:

$$\begin{aligned}\mathsf{N}(n) &\coloneqq \exists C_n.\ \mathsf{node}(n, C_n) * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma}\\ \mathsf{CSS}(n_1, n_2, C) &\coloneqq \boxed{\bullet(\mathsf{KS}, C)}^{\gamma} * (\exists b_1.\ \mathsf{lk}(n_1) \mapsto b_1 * (b_1\ ?\ \mathsf{True} : \mathsf{N}(n_1)))\\ &\quad * (\exists b_2.\ \mathsf{lk}(n_2) \mapsto b_2 * (b_2\ ?\ \mathsf{True} : \mathsf{N}(n_2)))\end{aligned}$$

Extending CSS to describe an unbounded number of nodes can be done using the iterated separating conjunction as (where r is the root node):

$$\mathsf{CSS}(r, C) \coloneqq \exists N.\ \boxed{\bullet(\mathsf{KS}, C)}^{\gamma} * \mathop{\Large *}_{n \in N} (\exists b.\ \mathsf{lk}(n) \mapsto b * (b\ ?\ \mathsf{True} : \mathsf{N}(n)))$$

This formula says there exists a set of nodes N, where each $n \in N$ has a lock location set to some boolean b and the resources in $\mathsf{N}(n)$. Extending the definition of N, on the other hand, is trickier. Recall that $\mathsf{ks}(n)$ was an abstract function, whose definition was to be provided by the implementation. For the two-node template, our example implementation used the following definition of the keyset:

$$\mathsf{ks}(n) \coloneqq \begin{cases} \{k \mid k \text{ is odd}\} & n = n_1 \\ \{k \mid k \text{ is even}\} & \text{otherwise} \end{cases}$$

Unfortunately, the keyset in most real-world implementations is a *global* quantity, i.e. $\mathsf{ks}(n)$ depends on nodes other than just n. For example, the rules of a B-tree dictate that the keyset of a node n depends on the keys in all the nodes on the path from the root to n. In Figure 2.1, the keyset of y_0 is $(-\infty, 4)$, and the keyset of y_2 is $[5, 8)$. This makes it impossible for us to define a function $\mathsf{ks}(n)$ that maps a B-tree node n to its keyset, because in Iris, a function can depend only on its arguments. We thus need a way to talk about a global quantity in a local manner.

5.2 LOCAL REASONING ABOUT GLOBAL PROPERTIES

As noted above, while separation logic is based on the concept of *local reasoning*, many important properties of data structure graphs depend on non-local information. For instance, we cannot express the property that a graph is a tree by conjoining per-node invariants. As we have seen above, we also cannot write down the keyset of a B-tree as a local function of each node. The flow framework [Krishna et al., 2018, 2020b] is a separation logic based approach that provides a mechanism to reason about global quantities in local proofs.

The flow framework [Krishna et al., 2020b] uses the concept of a *flow* – a function from nodes to values from some *flow domain* – to specify global graph properties in terms of node-local invariants. These flow values must satisfy the *flow equation*, i.e. they must be a fixpoint of a set of algebraic equations induced by the entire graph (thereby allowing one to capture global constraints at the node level). When modifying a graph, the framework allows one to perform a local proof that flow-based invariants are maintained via the notion of a *flow interface*. This is an abstraction of a graph region that specifies the flow values entering and exiting the region; if these are preserved then the flow values of the rest of the graph will be unchanged.

The rest of this section illustrates these concepts by considering the simple example of a linked-list data structure. We will come back to search structures in §5.4.

Suppose we have a graph G on a set of nodes N and we want to express the global property that it is a list rooted at some node r in terms of a condition on each node. To do this, we need to know some global information at each node: for instance, suppose there existed a function pc that mapped each node n to the number of paths from r to n.[1] If for every node n, $\mathsf{pc}(n) = 1$ and n has at most one outgoing edge (both node-local assertions) then we know that G must be a list rooted at r.

This path-counting function pc is an example of a flow because it can be defined as a solution to the flow equation:

$$\forall n \in N.\ \mathit{fl}(n) = \mathit{in}(n) + \sum_{n' \in N} e(n', n)(\mathit{fl}(n')) \qquad \text{(FlowEqn)}$$

This is a fixpoint equation on a function $\mathit{fl} \colon N \to M$, where M is a flow domain, in is an *inflow* that specifies the default/initial flow value of each node, and e is a mapping from pairs of nodes to *edge functions* that determine how the flow of one node affects the flow of its neighbor. The flow framework works with directed partial graphs that are augmented with a flow, called flow graphs. A flow graph is a tuple $H = (N, e, \mathit{fl})$ consisting of a finite set of nodes $N \subseteq \mathfrak{N}$ ($\mathfrak{N}$ is potentially infinite), a mapping from pairs of nodes to edge functions $e \colon N \times \mathfrak{N} \to E$, and a function fl such that (FlowEqn) is satisfied for some inflow in. Flow graph composition $H_1 \odot H_2$ is a partial operator that is a disjoint union of the nodes, edges, and flow values and is defined only if the resulting graph continues to satisfy (FlowEqn).

[1] We assume a definition of pc where $\mathsf{pc}(r) = 1$ even in acyclic graphs, because typically we are interested in the reachability of heap nodes from an external stack pointer.

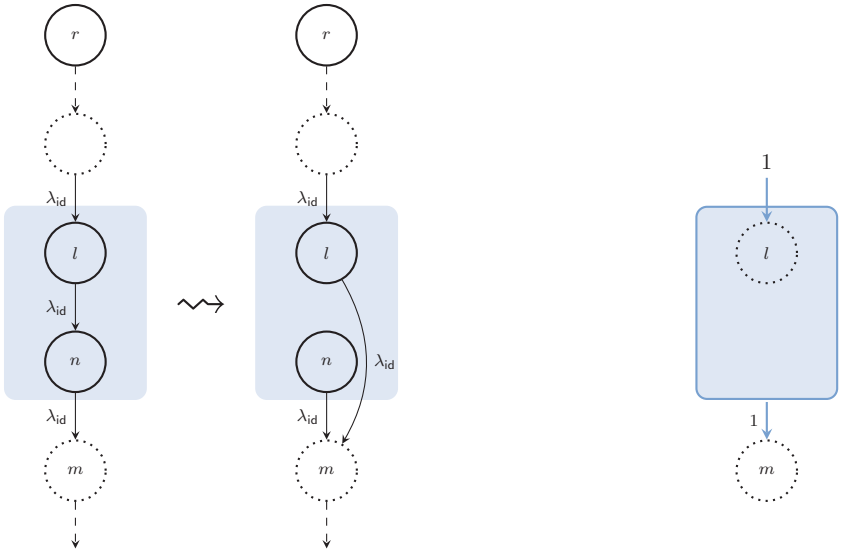

Figure 5.1: A flow-based view of the delete operation on a linked-list. The operation unlinks a node n from a linked-list by swinging the pointer from its predecessor l to its successor m. Edges are labeled with the path-counting flow domain (λ_0 edges omitted). The interface of the blue region $\{l, n\}$ is shown on the right, and is preserved by this update.

In the case of the path-counting flow, the flow domain M is $\mathbb{N}$, the inflow is $in(n) \coloneqq (n = r \ ? \ 1 : 0)$ (the root has inflow of 1 and all other nodes have inflow of 0), and the edge function $e(n, n')$ is the identity function $\lambda_{\mathsf{id}} \coloneqq (\lambda m.\ m)$ for all edges (n, n') in G and the zero function $\lambda_0 \coloneqq (\lambda m.\ 0)$ otherwise. The flow equation then reduces to the familiar constraint that the number of paths from r to n, $\mathsf{pc}(n)$, equals 1 if $n = r$ else 0, plus the sum of the number of paths to all n' that have an edge to n.

The problem with assuming that each node knows a flow value that satisfies some global constraint over the entire graph is that when a program modifies the graph, it can be hard to show that the flow-based invariants are maintained. In particular, when the program modifies a small part of the graph, say by modifying a single edge, we would like to prove that the flow invariants are preserved by reasoning only about a small region around the modified edge. The flow framework enables such local proofs by means of an abstraction of flow (sub)graphs called flow interfaces.

Consider the simple example of a singly-linked list deletion procedure that unlinks[2] a given node n from the list (Figure 5.1). The program swings the pointer from n's predecessor l to n's successor m. We use the path-counting flow and the flow-based local constraints described above to

[2]Recall from §3.3 that we assume a garbage-collected setting in this book.

express the invariant that the graph is a list (we show how to formally express this later). For a flow graph H over the path-counting flow domain, modifying a single edge (n, n') can potentially change the flow (the path-count) of every node reachable from n. However, notice that the modification shown in Figure 5.1 changes (l, n) to (l, m) where m is the successor of n. This preserves the flow of every node outside the modified subgraph $H_1 = H|_{\{l,n\}}$ (shown in blue in Figure 5.1) because there was one path coming out of H_1 and one path going to m both before and after the modification.

Flow interfaces build on this intuition; the interface $I = (in, out)$ of a flow graph H with domain N is a tuple consisting of the inflow $in\colon N \to M$ (e.g., how many incoming paths each node in H has) and the outflow $out\colon (\mathfrak{N} \setminus N) \to M$ (for our path-counting example, how many outgoing paths H has to each external node). Formally, the inflow of $H = (N, e, \mathit{fl})$ is the in that satisfies (FlowEqn) (this is unique [Krishna et al., 2020b]) and the outflow is defined as $out(n) \coloneqq \sum_{n' \in N} e(n', n)(\mathit{fl}(n'))$. For example, the flow interface of $\{l\}$ in the left of Figure 5.1 is $(\{l \rightarrowtail 1\}, \lambda_0[n \rightarrowtail 1])$ because l has one incoming path from outside $\{l\}$ and the subgraph $\{l\}$ has one outgoing path to n. The interface of $\{l, n\}$ in the left and center of Figure 5.1 is $(\{l \rightarrowtail 1, n \rightarrowtail 0\}, \lambda_0[m \rightarrowtail 1])$, which is depicted abstractly on the right. The flow framework tells us that if we have $H = H_1 \odot H_2$ and we modify H_1 to some H_1' with the same interface, then $H' = H_1' \odot H_2$ exists. This means that the flow of all nodes in H_2 is unchanged; thus it suffices to check that H_1' satisfies the flow-based invariant and has the same interface as H_1, which are both local checks.

One can capture any graph property of interest by instantiating the flow domain appropriately [Krishna et al., 2020b]. This involves specifying the desired property in terms of two constraints: a local constraint on the flow of every node, and a global constraint on the inflow and outflow of the entire data structure (the latter is typically used to identify root nodes). We will demonstrate this §5.4 when we encode keysets using flows. First, we show how to perform flow-based reasoning in Iris.

5.3 THE FLOW INTERFACE RA

This section formally defines the notions presented in the previous section, in particular, a flow interface resource algebra that will allow us to perform flow-based reasoning in Iris.

Definition 5.1 Flow Domain. A *flow domain* $(M, +, 0, E)$ consists of a commutative cancellative (total) monoid $(M, 0, +)$ and a set of functions $E \subseteq M \to M$.

Example 5.2 The flow domain used for the path-counting flow is $(\mathbb{N}, +, 0, \{\lambda_{\mathsf{id}}, \lambda_0\})$, consisting of the monoid on natural numbers under addition and the set of edge functions containing only the identity function and the zero function.

Definition 5.3 Graph. A *(partial) graph* $G = (N, e)$ consists of a finite set of nodes $N \subseteq \mathfrak{N}$ and a mapping from pairs of nodes to edge functions $e\colon N \times \mathfrak{N} \to E$.

A *flow of graph* $G = (N, e)$ *under inflow* $in \colon N \to M$ is a solution of the following fixpoint equation (the same as (FlowEqn), repeated for clarity) over G, denoted $\mathsf{FlowEqn}(in, e, \mathit{fl})$:

$$\forall n \in \mathsf{dom}(in).\ \mathit{fl}(n) = in(n) + \sum_{n' \in \mathsf{dom}(in)} e(n', n)(\mathit{fl}(n'))$$

Note that every graph need not have a flow; there are graphs (N, e) and inflows in for which $\mathsf{FlowEqn}(in, e, \mathit{fl})$ has no solution. On the other hand, graphs that have a flow are called flow graphs.

Definition 5.4 Flow Graph. A *flow graph* $H = (N, e, \mathit{fl})$ consists of a graph (N, e) and a function $\mathit{fl} \colon N \to M$ such that there exists an *inflow* $in \colon N \to M$ satisfying $\mathsf{FlowEqn}(in, e, \mathit{fl})$.

We let $\mathsf{dom}(H) = N$, and sometimes identify H and $\mathsf{dom}(H)$ to ease notational burden. Two flow graphs with disjoint domains always compose to a graph, but this will only be a flow graph if their flows are chosen consistently to admit a solution to the resulting flow equation (thus, the flow graph algebra below has a special *invalid* element $H_\lightning$, which is the result of flow graph composition of incompatible flow graphs).

Definition 5.5 Flow Graph Algebra. The *flow graph algebra* $(\mathsf{FG}, \odot, H_\emptyset)$ for flow domain $(M, +, 0, E)$ is defined by

$$\begin{aligned}
\mathsf{FG} &::= H \in \{(N, e, \mathit{fl}) \mid (N, e, \mathit{fl}) \text{ is a flow graph}\} \mid H_\lightning \\
(N_1, e_1, \mathit{fl}_1) \odot (N_2, e_2, \mathit{fl}_2) &:= \begin{cases} H & H = (N_1 \uplus N_2, e_1 \uplus e_2, \mathit{fl}_1 \uplus \mathit{fl}_2) \in \mathsf{FG} \\ H_\lightning & \text{otherwise} \end{cases} \\
_ \odot H_\lightning &:= H_\lightning \odot _ := H_\lightning \\
H_\emptyset &:= (e_\emptyset, \mathit{fl}_\emptyset)
\end{aligned}$$

where $e_\emptyset$ and $\mathit{fl}_\emptyset$ are the edge functions and flow on the empty set of nodes $N = \emptyset$.

Cancellativity of the flow domain operator $+$ is key to defining an abstraction of flow graphs that permits local reasoning. The following lemma follows from the fact that $+$ is cancellative.

Lemma 5.6 *Given a flow graph* $(N, e, \mathit{fl}) \in \mathsf{FG}$*, there exists a unique inflow* $in \colon N \to M$ *such that* $\mathsf{FlowEqn}(in, e, \mathit{fl})$.

Proof. Suppose in and in' are two solutions to $\mathsf{FlowEqn}(\_, e, \mathit{fl})$. Then, for any n,

$$\mathit{fl}(n) = in(n) + \sum_{n' \in \mathsf{dom}(in)} e(n', n)(\mathit{fl}(n')) = in'(n) + \sum_{n' \in \mathsf{dom}(in')} e(n', n)(\mathit{fl}(n'))$$

which, by cancellativity of the flow domain, implies that $in(n) = in'(n)$. □

Our abstraction of flow graphs consists of two complementary notions. Lemma 5.6 implies that any flow graph has a unique inflow. Thus we can define an inflow function that maps each flow graph $H = (N, e, \mathit{fl})$ to the unique inflow $\mathsf{inf}(H)\colon H \to M$ such that $\mathsf{FlowEqn}(\mathsf{inf}(H), e, \mathit{fl})$. We can also define the *outflow* of H as the function $\mathsf{outf}(H)\colon \mathfrak{N} \setminus N \to M$ defined by

$$\mathsf{outf}(H)(n) \coloneqq \sum_{n' \in N} e(n', n)(\mathit{fl}(n')).$$

Definition 5.7 Flow Interface. Given a flow graph $H \in \mathsf{FG}$, its *flow interface* $\mathsf{int}(H)$ is the tuple $(\mathsf{inf}(H), \mathsf{outf}(H))$ consisting of its inflow and its outflow.

We use $I.\mathit{in}$ and $I.\mathit{out}$ to denote, respectively, the inflow and outflow of an interface I. We can finally define the flow interface resource algebra, which will allow us to use flow interfaces in our Iris proofs.

Definition 5.8 Flow Interface Algebra. The flow interface algebra $(\mathsf{FI}, \overline{\mathcal{V}}, |-|, \oplus)$ is defined by

$$\mathsf{FI} ::= I \in \{\mathsf{int}(H) \mid H \in \mathsf{FG}\} \mid I_\lightning \qquad \overline{\mathcal{V}}(a) \coloneqq a \neq I_\lightning \qquad |a| \coloneqq I_\emptyset$$

$$I_1 \oplus I_2 \coloneqq \begin{cases} \mathsf{int}(H) & \exists H_1, H_2.\ H = H_1 \odot H_2 \land \forall i \in \{1,2\}\,.\ \mathsf{int}(H_i) = I_i \\ I_\lightning & \text{otherwise,} \end{cases}$$

where $I_\emptyset \coloneqq \mathsf{int}(H_\emptyset)$.

Theorem 5.9 *The flow interface algebra $(\mathsf{FI}, \overline{\mathcal{V}}, |-|, \oplus)$ is a resource algebra.*

5.4 ENCODING KEYSETS USING FLOWS

We now answer the question of how to define a keyset function in a node-local way, using flow interfaces.

To define keysets using flows, we build on the concept of edgesets. Recall that the edgeset $\mathsf{es}(n, n')$ is the set of keys for which an operation arriving at a node n traverses (n, n'). This is a node-local concept, and hence can be expressed in Iris. Let the *inset* of a node n, written $\mathsf{ins}(n)$, be defined by the following fixpoint equation

$$\forall n \in N.\ \mathsf{ins}(n) = \mathit{in}(n) \cup \bigcup_{n' \in N} \mathsf{es}(n', n) \cap \mathsf{ins}(n')$$

where $\mathit{in}(n) \coloneqq (n = r\ ?\ \mathsf{KS} : \emptyset)$. The inset of a node n is thus KS if n equals the root r, else the set of keys k that are in the inset of a predecessor n' such that $k \in \mathsf{es}(n', n)$. Intuitively, $\mathsf{ins}(n)$ is the

$$\mathsf{ins}(I, n) \coloneqq I.in(n)$$

$$\mathsf{outs}(I, n') \coloneqq I.out(n') \qquad \mathsf{outs}(I) \coloneqq \bigcup_{n' \notin \mathsf{dom}(I)} \mathsf{outs}(I, n')$$

$$\mathsf{ks}(I, n) \coloneqq \mathsf{ins}(I, n) \setminus \mathsf{outs}(I, n)$$

$$\mathsf{N}(n, C_n) \coloneqq \exists I_n.\ \mathsf{node}(n, I_n, C_n) * \boxed{\circ(\mathsf{ks}(I_n, n), C_n)}^{\gamma_k} * \boxed{\circ I_n}^{\gamma_I} * \mathsf{dom}(I_n) = \{n\}$$

$$\varphi(r, I) \coloneqq \overline{\mathcal{V}}(I) \land I.in(r) = \mathsf{KS} \land I.out = \lambda_0$$

$$\mathsf{CSS}(r, C) \coloneqq \exists I.\ \boxed{\bullet I}^{\gamma_I} * \varphi(r, I) * \boxed{\bullet(\mathsf{KS}, C)}^{\gamma_k} * \mathop{\Large *}_{n \in \mathsf{dom}(I)} \Big(\exists b.\ \mathsf{lk}(n) \mapsto b * (b\ ?\ \mathsf{True} : \exists C_n.\ \mathsf{N}(n, C_n)) \Big)$$

Figure 5.2: The definition of CSS and other predicates using a flow-based encoding of keysets.

set of keys for which operations could potentially arrive at n in a sequential setting. For example, in Figure 2.1 insets are shown in the top-left of each node; $\mathsf{ins}(y_2) = [5, 8)$ and $\mathsf{ins}(n') = [5, \infty)$. Let the *outset* of n, $\mathsf{outs}(n)$, be the keys in the union of edgesets of edges leaving n. The keyset can then be defined as $\mathsf{ks}(n) = \mathsf{ins}(n) \setminus \mathsf{outs}(n)$.

If the equation defining the inset looks familiar, the reason is that it is just (FlowEqn) using sets and set operations, and edge functions that take the intersection with the appropriate edgeset. This means we can define a flow domain where the flow at each node is the inset of that node. This will allow us to talk about the keyset in terms of node-local conditions: in particular, we can now use the keyset in the keyset RA ghost state described in §4.5.

Encoding the inset as a flow requires using multisets of keys[3] as the flow domain. We label each edge (n, n') in a graph G by the function $e_{\mathsf{es}(n,n')} \coloneqq (\lambda X.\ \mathsf{es}(n, n') \cap X)$. If the global inflow is $in = (\lambda n.\ (n = r\ ?\ \mathsf{KS} : \emptyset))$, which encodes the fact that operations on all keys k start at the root r, then the flow equation implies that $\mathit{fl}(n)$ is the inset of n.

As with the keyset RA, we use an authoritative version of the flow interface RA (Auth(FI), see Definition 4.4) in our proofs. This enables us to reason about the concurrent setting where threads can lock nodes and take them out of the shared state. Since we are now using two kinds of ghost state, we use γ_I to denote the Auth(FI) ghost location and γ_k for the Auth(Keyset) ghost location.

[3] We cannot use sets of keys because a flow domain is a cancellative commutative monoid (Definition 5.1), and unlike multiset union, set union is not cancellative.

The resulting definitions are shown in Figure 5.2. The inset, outset, and keyset are defined in terms of a flow interface as described above (we overload the same symbols used before because they express the same quantities). The node predicate $\mathsf{N}(n, C_n)$ now has a fragment of both the keyset RA and flow interface RA corresponding to the node n. Note that n's keyset is now a function of n's singleton interface and n, making it local as desired. The predicate φ contains the constraints on the interface of the global search structure which are needed to make the flow of each node be interpreted as the inset of each node. CSS contains the authoritative global interface $\boxed{\bullet I}^{\gamma_I}$, as well as the authoritative keyset RA element as before.

With such a definition, we can reason about search structures with an unbounded number of nodes. In the next chapter we will see how to prove three templates for real-world search structures.

CHAPTER 6

Verifying Single-copy Concurrent Templates

This chapter shows how to bring together all the concepts developed so far in order to verify template algorithms for concurrent search structures in which each key is present at most once in a data structure. The three templates we describe are based on three common concurrency techniques, and were first described in template form by Shasha and Goodman [1988].

$$\left\langle\, C.\ \mathsf{CSS}(r, C) \,\right\rangle\ \texttt{cssOp}\ \omega\ k\ \left\langle\, \mathsf{res}.\ \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', \mathsf{res}) \,\right\rangle$$

$$\Psi_\omega(k, C, C', \mathsf{res}) := \begin{cases} C' = C \wedge (\mathsf{res} \iff k \in C) & \omega = \texttt{search} \\ C' = C \cup \{k\} \wedge (\mathsf{res} \iff k \notin C) & \omega = \texttt{insert} \\ C' = C \setminus \{k\} \wedge (\mathsf{res} \iff k \in C) & \omega = \texttt{delete} \end{cases}$$

Figure 6.1: The atomic specification of core search structure operations. We prove that all search structure templates in this book satisfy this specification.

Remember, our aim is to prove the atomic specification shown in Figure 6.1 for the template method `cssOp` that represents, via the parameter ω, an arbitrary search structure operation (either search, insert, or delete). This specification uses an abstract predicate $\mathsf{CSS}(r, C)$ that represents a search structure with root r containing the set of keys C. The binder on C in the precondition is a special *pseudo-quantifier* that captures the fact that during the execution of ω, the value of C can change (e.g. by concurrent operations) but at the linearization point, ω on operation key k changes $\mathsf{CSS}(r, C)$ to $\mathsf{CSS}(r, C')$ in an atomic step. The new set of keys C', and the eventual return value res, satisfy the predicate $\Psi_\omega(k, C, C', \mathsf{res})$ – here C is bound to the contents *just before* the linearization point. The bottom line is that clients of the search structure can pretend that they are using a sequential implementation with specification Ψ_ω.

$$\mathsf{inFP}(n) \mathrel{-\!\!*} \left\langle\, C.\ \mathsf{CSS}(r, C) \,\right\rangle \texttt{lockNode}\ x \left\langle\, \mathsf{CSS}(r, C) * \mathsf{N}(n, I_n, C_n) \,\right\rangle$$

$$\mathsf{N}(n, I_n, C_n) \mathrel{-\!\!*} \left\langle\, C.\ \mathsf{CSS}(r, C) \,\right\rangle \texttt{unlockNode}\ \mathsf{n} \left\langle\, \mathsf{CSS}(r, C) \,\right\rangle$$

Figure 6.2: High-level specifications for the lock module used by all template proofs in this chapter. These can be proved from the low-level specifications we proved in Chapter 3 and the definitions of CSS and inFP.

All the template proofs in this chapter assume some immplementation of `lockNode` and `unlockNode` that satisfy the following specifications:[1]

$$\left\langle\, b.\ \mathsf{lk}(x) \mapsto b \,\right\rangle \texttt{lockNode}\ x \left\langle\, \mathsf{lk}(x) \mapsto \mathsf{True} * b = \mathsf{False} \,\right\rangle$$

$$\left\langle\, \mathsf{lk}(x) \mapsto \mathsf{True} \,\right\rangle \texttt{unlockNode}\ x \left\langle\, \mathsf{lk}(x) \mapsto \mathsf{False} \,\right\rangle$$

To simplify the upcoming proofs, we will assume that `lockNode` and `unlockNode` satisfy the higher-level specifications shown in Figure 6.2, which can be proved easily from the above specifications and the definitions of CSS and inFP.

These specifications are written in a new form, using the magic wand $\mathrel{-\!\!*}$, which is the ownership analogue of implication (see Chapter 3 for more details). The specification of `lockNode` says that if we own $\mathsf{inFP}(n)$, i.e. if we know that n is in the footprint of the structure, then we can use the atomic triple that follows. The triple tells us that `lockNode` operates on the search structure $\mathsf{CSS}(r, C)$, and removes the node $\mathsf{N}(n, I_n, C_n)$ from it (allowing the caller to move it to its local state) while returning the search structure unmodified.

Similarly, the specification of `unlockNode` says that if a thread owns the node predicate for n, then it can call `unlockNode` in an atomic step to put n back into the shared search structure.

6.1 THE GIVE-UP TEMPLATE

We start by considering a template algorithm that uses the well-known *give-up* style of concurrency (Figure 6.3). The proof of this template is very similar to that of the link template, but simpler, hence we describe it first.

The give-up template, like the link template, uses locks only when reading or writing from a node and does not hold locks while traversing from one node to the next. Unlike the link template, there are no link edges added by threads that move data from one node to another. Instead, each node stores a *range* field: this is an under-approximation of that node's inset. Upon arriving at a new node n, each thread locks the node and checks its query key k against the range of n. If k is in the range of n then the thread knows that it is still on the correct path, and it continues. If not, it gives up: it

[1] We showed in Chapter 3 that a simple spin-lock implementation satisfies these specifications. However, note that one can use more complex lock implementations, as long as they satisfy these specifications.

```
1  let rec traverse r n k =
2    lockNode n;
3    if inRange n k then
4      match findNext n k with
5      | None -> n
6      | Some n' -> unlockNode n;
7          traverse n' k
8    else
9      unlockNode n;
10     traverse r r k
```

```
11 let rec cssOp ω r k =
12   let n = traverse r r k in
13   match decisiveOp ω n k with
14   | None ->
15       unlockNode n;
16       cssOp ω r k
17   | Some res ->
18       unlockNode n;
19       res
```

Figure 6.3: The give-up template algorithm. The `cssOp` method is the main method, and represents all the core search structure operations via the parameter ω. It makes use of an auxiliary method `traverse` that recursively traverses the search structure until it finds the node upon which to operate.

relinquishes the lock on n and goes back to the root of the data structure to retry. (In fact, it could go back to any previous node to retry, but eventually it might have to go back to the root. For simplicity we consider the version of the algorithm that goes back to the root immediately.)

The code for this template algorithm is given in Figure 6.3. Note that in addition to the helper functions `findNext` and `decisiveOp`, this template assumes a helper function `inRange`. When called as `inRange` $n\,k$, this function returns `true` if and only if k is in the range of n.

The give-up template can be instantiated by a B+ tree, for instance, by adding two additional fields to each node n. These fields keep track of lower and upper bounds for keys that are present in the subtree rooted at n. When a thread looking for k arrives at a node n, it checks if k is in the range of n (by checking if k is between the lower and upper bounds inclusively), and gives up and restarts if not. Though we have conceived a range as consisting of a lower and upper bound, in fact a range can be an arbitrary function as long as it is a subset of the inset. For example, it can be a set of key values that hash to a particular value for a hash table.

6.1.1 PROOF OF THE GIVE-UP TEMPLATE

The definition of the search structure predicate CSS for the give-up template is given in Figure 6.4. This is almost the same as the one we developed in the last chapter using flows (see Figure 5.2 in §5.4), but with an extra form of ghost state that is used to reason about traversals. The reason we need additional ghost state is that the give-up template performs a traversal over the search structure where it holds no locks when moving from one node to the next. For instance, at the point when the `traverse` method is called, the node n is not locked. Yet, to be able to apply the specification of `lockNode`, we need to know that n is a node in the data structure (and not some arbitrary memory address that may not be allocated).

We solved this issue in the two-node template proof by defining a predicate $\mathsf{inFP}(n_1, n_2, n)$, that asserted that n is one of the two nodes $\{n_1, n_2\}$ in the structure. This approach no longer works

$$\mathsf{ins}(I, n) \coloneqq I.in(n)$$

$$\mathsf{outs}(I, n') \coloneqq I.out(n') \qquad \mathsf{outs}(I) \coloneqq \bigcup_{n' \notin \mathsf{dom}(I)} \mathsf{outs}(I, n')$$

$$\mathsf{ks}(I, n) \coloneqq \mathsf{ins}(I, n) \setminus \mathsf{outs}(I, n)$$

$$\mathsf{N}(n, C_n) \coloneqq \exists I_n.\ \mathsf{node}(n, I_n, C_n) * \boxed{\circ(\mathsf{ks}(I_n, n), C_n)}^{\gamma_k} * \boxed{\circ I_n}^{\gamma_I} * \mathsf{dom}(I_n) = \{n\}$$

$$\varphi(r, I) \coloneqq \overline{\mathcal{V}}(I) \land I.in(r) = \mathsf{KS} \land I.out = \lambda_0$$

$$\mathsf{CSS}(r, C) \coloneqq \exists I.\ \boxed{\bullet I}^{\gamma_I} * \varphi(r, I) * \boxed{\bullet(\mathsf{KS}, C)}^{\gamma_k} * \boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f} * \mathop{\Large *}_{n \in \mathsf{dom}(I)} \Big(\exists b.\ \mathsf{lk}(n) \mapsto b * (b\ ?\ \mathsf{True} : \exists C_n.\ \mathsf{N}(n, C_n))\Big)$$

Figure 6.4: The definition of CSS and other predicates used by the give-up template proof.

since we are in the general case where the structure has an unbounded number of nodes that are not known to us a priori.

Our solution is to use an authoritative RA of sets of nodes, $\textsc{Auth}(\mathfrak{N})$, at a new ghost location γ_f. The authoritative element owned by CSS is $\boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f}$, and this captures the domain of the shared state (which is equal to the domain of the global flow interface). The following properties of $\textsc{Auth}(\mathfrak{N})$ allow threads to take snapshots of the footprint and assert locally that a given node is in the footprint:

$$\textsc{auth-set-upd}\ \frac{X \subseteq Y}{\bullet X \rightsquigarrow \bullet Y} \qquad \textsc{auth-set-snap}\ \bullet X \rightsquigarrow \bullet X \cdot \circ X \qquad \textsc{auth-set-valid}\ \frac{\overline{\mathcal{V}}(\bullet X \cdot \circ Y)}{Y \subseteq X}$$

We then define the footprint predicate as

$$\mathsf{inFP}(n) \coloneqq \boxed{\circ\, \{n\}}^{\gamma_f}$$

which expresses ownership of a fragment $\{n\}$ of the domain. By AUTH-SET-VALID, this along with $\boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f}$ implies that $n \in \mathsf{dom}(I)$. Threads can thus use AUTH-SET-SNAP and AUTH-FRAG-OP to create the resource $\mathsf{inFP}(n)$, which can then be moved into the thread's local state.

Before we move on to the proof of the give-up template, let us review the assumptions made by the give-up template on implementations (Figure 6.5). As usual, they are all Hoare triples that

$\{\mathsf{node}(n, I_n, C_n)\}$
`inRange n k`
$\{v.\ \mathsf{node}(n, I_n, C_n) * (v = \mathsf{True} \Rightarrow k \in \mathsf{ins}(I_n, n))\}$

$\{\mathsf{node}(n, I_n, C_n) * k \in \mathsf{ins}(I_n, n)\}$
`findNext n k`
$\left\{v.\ \mathsf{node}(n, I_n, C_n) * \left(\begin{array}{l} v = \mathsf{None} * k \notin \mathsf{outs}(I_n) \\ \qquad \vee\ v = \mathsf{Some}(n') * k \in \mathsf{outs}(I_n, n')\end{array}\right)\right\}$

$\{\mathsf{node}(n, I_n, C_n) * k \in \mathsf{ks}(I_n, n)\}$
`decisiveOp` ω `n k`
$\left\{v.\ \mathsf{node}(n, I_n, C'_n) * \left(\begin{array}{l} v = \mathsf{None} * C_n = C'_n \\ \qquad \vee\ v = \mathsf{Some}(v') * \Psi_\omega(k, C_n, C'_n, v')\end{array}\right)\right\}$

$\mathsf{node}(n, I_n, C_n) * \mathsf{node}(n, I'_n, C'_n) \mathrel{-\!\!*} \mathsf{False}$

Figure 6.5: The assumptions made by the give-up template on implementations.

operate on the abstract node predicate, meaning the implementation can be sequential. We require that `inRange n k` return a boolean value, which if true implies that k is in the inset of n. (If it is false, we do not require any additional information, because the algorithm gives up and restarts from the root.) The `findNext` method is used by the traversal at each step: given a node n and a key k, it either returns None, indicating that there is no outgoing edge with k in its edgeset, or returns $\mathsf{Some}(n')$ such that k is in the edgeset of (n, n'). Neither of these two methods modify the given node n. The `decisiveOp` method has a similar spec to what we have used before, except that now we allow it to also fail: if it returns None then we require it to return the node n with unmodified contents, while if it returns $\mathsf{Some}(v')$ then we require it to satisfy the per-operation specification $\Psi_\omega(k, C_n, C'_n, v')$ as before.

Moving on to the proofs, we prove the following specification for the `traverse` helper function:

$$\mathsf{inFP}(n) \mathrel{-\!\!*} \quad \langle\, C.\ \mathsf{CSS}(r, C) \,\rangle \ \texttt{traverse r n k}\ \langle\, v.\ \mathsf{CSS}(r, C) * \mathsf{N}(v, I_v, C_v) * k \in \mathsf{ks}(I_v, v) \,\rangle$$

This specification says that if n is in the footprint of the search structure, then `traverse` operates on the entire search structure and returns a locked node v such that k is in the keyset of v, while leaving the global contents unmodified.

The proof of `traverse` is shown in Figure 6.6. Note that the $\mathsf{inFP}(n)$ predicate from the precondition is available in our proof context at the beginning of the proof. Hence we can use it to apply the high-level `lockNode` specification (Figure 6.2) to get the node $\mathsf{N}(n, I_n, C_n)$ into the thread-local state. We then use the specification of `inRange` from Figure 6.5.

```
1  inFP(n) -*⟨ C. CSS(r, C) ⟩
2  let rec traverse r n k =
3    {inFP(n)}
4    lockNode n;
5    {N(n, I_n, C_n)}
6    if inRange n k then
7      {N(n, I_n, C_n) * k ∈ ins(I_n, n)}
8      match findNext n k with
9      | None -> {N(n, I_n, C_n) * k ∈ ins(I_n, n) * k ∉ outs(I_n)}
10       n
11     | Some n' -> {N(n, I_n, C_n) * k ∈ ins(I_n, n) * k ∈ outs(I_n, n')}
12       {N(n, I_n, C_n) * inFP(n')}
13       unlockNode n; {inFP(n')}
14       traverse n' k
15   else
16     {N(n, I_n, C_n)}
17     unlockNode n;
18     {inFP(r)}
19     traverse r r k
20 ⟨ v. CSS(r, C) * N(v, I_v, C_v) * k ∈ ks(I_v, v) ⟩
```

Figure 6.6: The proof of the `traverse` method of the give-up template algorithm.

In the then branch, this gives us the additional predicate $k \in \mathsf{ins}(I_n, n)$. We use this to apply the specification of `findNext`, which leads to two cases. In the case where `findNext` returns None, the specification tells us that we have $k \in \mathsf{ins}(I_n, n) * k \notin \mathsf{outs}(I_n)$. By the definition of ks in Figure 6.4, this is exactly $k \in \mathsf{ks}(I_n, n)$, which allows us to prove the postcondition (technically, this is the linearization point, and we use the appropriate proof rules to "commit" the change to the shared structure and establish the postcondition).

If `findNext` returns $\mathsf{Some}(n')$, then we have $k \in \mathsf{ins}(I_n, n) * k \in \mathsf{outs}(I_n, n')$ from `findNext`'s postcondition. Before we move on to the unlocking and give away the node predicate, note that we need to prove $\mathsf{inFP}(n')$ in order to satisfy the precondition of the recursive call to `traverse`. We obtain this by opening the precondition and using $k \in \mathsf{outs}(I_n, n')$ along with $\varphi(I)$. Since I_n is a sub-interface of I, and $\varphi(I)$ specifies that I has no outflow, n can have an outflow of k to n' only if n' is also a node in I. We thus obtain $\mathsf{inFP}(n')$, and this completes this branch.

In the else branch of the call to `inRange`, we simply use the high-level specification of `unlockNode` to give back the node predicate. We can then use induction to assume that the recursive call to `traverse` will give us the desired postcondition. The precondition of the recursive call requires us to prove $\mathsf{inFP}(r)$, but note that this is always true from the definition of CSS (as $\varphi(r, I) \Rightarrow r \in \mathsf{dom}(I)$). This completes the proof of `traverse`.

1 $\langle\, C.\ \mathsf{CSS}(r, C) \,\rangle$
2 **let rec** cssOp ω r k =
3 $\{\mathsf{inFP}(r)\}$
4 **let** n = traverse r r k **in**
5 $\{\mathsf{N}(n, I_n, C_n) * k \in \mathsf{ks}(I_n, k)\}$
6 **match** decisiveOp ω n k **with**
7 | None -> $\{\mathsf{N}(n, I_n, C_n)\}$
8 unlockNode n; $\{\mathsf{True}\}$
9 cssOp ω r k
10 | Some res ->
11 $\{\mathsf{node}(n, I_n, C'_n) * \Psi_\omega(k, C_n, C'_n, \mathsf{res}) * k \in \mathsf{ks}(I_n, n) * \cdots\}$
12 (* Linearization point: open precondition *)
13 $\left\langle \boxed{\circ(\mathsf{ks}(I_n, n), C_n)}^{\gamma_k} * \Psi_\omega(k, C_n, C'_n, \mathsf{res}) * k \in \mathsf{ks}(I_n, n) * \boxed{\bullet(\mathsf{KS}, C)}^{\gamma_k} * \cdots \right\rangle$
14 $\left\langle \boxed{\circ(\mathsf{ks}(I_n, n), C'_n)}^{\gamma_k} * \Psi_\omega(k, C, C', \mathsf{res}) * \boxed{\bullet(\mathsf{KS}, C')}^{\gamma_k} * \cdots \right\rangle$ (* By KS-UPD *)
15 $\langle\, \mathsf{N}(n, I_n, C'_n) * \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', \mathsf{res}) \,\rangle$
16 unlockNode n;
17 $\langle\, \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', \mathsf{res}) \,\rangle$ (* Prove postcondition, commit *)
18 $\{\mathsf{True}\}$
19 res
20 $\langle\, v.\ \mathsf{CSS}(r, C') * \Psi_\omega(k, C, C', v) \,\rangle$

Figure 6.7: The proof of the give-up template algorithm.

We can now describe the proof of cssOp. As mentioned above, the definition of CSS gives us $\mathsf{inFP}(r)$, so we open the precondition to get $\mathsf{inFP}(r)$ and use it to apply the specification of traverse. This gives us a state where we have a node n such that k is in the keyset of n, which is all we need to apply the specification of decisiveOp. In the case where decisiveOp fails and returns None, we simply use the high-level specification of unlockNode to give back the node n and retry (using induction to handle the recursive call).

On the other hand, if decisiveOp succeeds, then we have the state shown in line 11 (we write $\cdots$ to hide the resources irrelevant to the next step in the proof). Note that we cannot yet write $\mathsf{N}(n, I_n, C'_n)$ because decisiveOp modified only node and not the rest of the ghost state in N. We update this ghost state as well as the ghost state in the shared state in the next step. Since the call to unlockNode is the linearization point, we open the precondition and prepare to commit our changes to the shared state. In particular, we focus on the keyset RA ghost state, as shown in line 13. We now have all the resources needed to use the rule KS-UPD and update both the contents of n in its fragment from C_n to C'_n and the global contents from C to some C' such that $\Psi_\omega(k, C, C', \mathsf{res})$ holds. The state that we have now has all the ghost state updated appropriately, so we can fold the node n as $\mathsf{N}(n, I_n, C'_n)$. We can then use the specification of unlockNode to give the node n back to the shared state, and prove the postcondition of cssOp in order to finish the proof.

6.1.2 MAINTENANCE OPERATIONS.

The template algorithms we have seen so far cover the core search structure operations search, insert, and delete. However, real search structures also need maintenance operations in order to function correctly. For example, in a B+ tree, successive inserts of nearby keys can make a leaf node full, necessitating a split operation to split the leaf into two. Conversely, practical B+-tree implementations also have a free-at-empty operation when a node becomes empty. (Classically, B+-trees use "merges" when neighboring nodes become less than half-full. However, free-at-empty turns out to be more efficient, because merged nodes tend to be split soon afterwards if, as is common, there are more inserts than deletes to the structure.) To ensure that these maintenance operations do not invalidate any core operations, we must verify that they (a) preserve the invariants of the search structure, and (b) do not change the contents of the entire structure.

Both these conditions can be expressed in the following specification for an arbitrary maintenance operation `maintOp`:

$$\langle\, C.\ \mathsf{CSS}(r, C) \,\rangle\ \texttt{maintOp r}\ \langle\, \mathsf{CSS}(r, C) \,\rangle$$

This triple gives the maintenance operation the permission to modify the search structure $\mathsf{CSS}(r, C)$ as long as it appears, to any other thread, to instantaneously modify the structure to another valid search structure $\mathsf{CSS}(r, C)$ with the same contents. If the maintenance operation operates on locked nodes, then its proof can again be split between a concurrent proof of the `maintOp` and a sequential proof of the operations it performs on locked nodes. We omit these proofs here, but they can be proved using the same techniques that we have shown so far.

For the give-up template, preserving CSS includes preserving the invariant that the range of every node is a subset of its inset. In the B+ tree example, the range of the root at any given time is the entire key space KS. The range of a non-root node n is set to its inset when n is created. If n is later split or merged, then its range is reduced to the inset it will be at the end of the split or merge. A thread searching for k might have visited the parent of n before the split began and then might visit node n after n was split. The thread would then see that the key k is not in the range of n and give up.

The split operation involves creating a new node, and to show that this does not invalidate any of the flow-based invariants, we use the following notion.[2]

Definition 6.1 A flow interface I' is a *domain extension* of interface I, written $I \Subset I'$, if and only if

1. $\mathsf{dom}(I) \subseteq \mathsf{dom}(I')$,
2. $\forall n \in I.\ I.in(n) = I'.in(n)$, and
3. $\forall n' \notin I'.\ I.out(n') = I'.out(n')$.

[2]Remember, we write I for $\mathsf{dom}(I) = \mathsf{dom}(I.in)$ when it is clear from context.

This definition allows I' to differ from I by having a larger domain, as long as the new nodes are fresh and edges from the new nodes do not change the outflow of the modified region. We can show that replacing an interface with a domain extension is a frame-preserving update in the flow interface RA:

$$\text{FLOWINT-DOM-UPD}\quad \frac{I_1 \Subset I_1' \qquad I_1' \cap I_2 = \emptyset \qquad \forall n \in I_1' \setminus I_1.\ I_2.out(n) = 0}{(\bullet I_1 \oplus I_2, \circ I_1) \rightsquigarrow \{(\bullet I_1' \oplus I_2, \circ I_1') \mid (I_1 \oplus I_2) \Subset (I_1' \oplus I_2)\}}$$

We use this lemma in the proof of the split operation to update the flow interface ghost state when adding the new node.

6.2 THE LINK TEMPLATE

The link template is very similar to the give-up template, and shares much of its proof. While the link template algorithm might even look simpler than the give-up, for it does not have the `inRange` helper function, it is in fact a bit more complicated to prove. This section explains why, and then shows how to extend the proof of the give-up template to the link template.

6.2.1 INREACH

The main reason the give-up proof does not work for the link template is that without `inRange` it is hard to prove that the link template's traversal stays on track and finds the correct node (i.e. a node n with $k \in \mathsf{ks}(n)$) at the end.

In the absence of concurrent operations (particularly concurrent split operations), this follows because we start off at the root r, where by definition $k \in \mathsf{ins}(r)$, and traverse an edge (n, n') only when $k \in \mathsf{es}(n, n')$, maintaining the invariant that $k \in \mathsf{ins}(n)$. When a node n has no outgoing edge having k in its edgeset, we know by definition that $k \in \mathsf{ks}(n)$.

In the presence of concurrent split operations, the $k \in \mathsf{ins}(n)$ invariant no longer holds because the inset of a node n shrinks after a split. For example, Figure 6.8 shows a B-link tree state in between the half-split and full-split. When the full-split completes and r is linked to n' (Figure 6.9), then the inset of n will be reduced from $(-\infty, \infty)$ to $(-\infty, 5)$ as all keys larger than 5 will go from r directly to n'. This means that an operation looking for a key $k > 5$ which was at n before the split will now find itself at a node such that $k \notin \mathsf{ins}(n)$.

Fortunately, the operation is not lost: if it traverses the link edge, it will arrive at a node with k in its inset (namely, n'). This means that if we add k back to the inset of n, then we would not be changing the keyset of any node: k will not be in n's keyset as it is in the edgeset of the link edge, and k is already in the inset of n'. Because this quantity is no longer the inset (as k would not arrive at n in a sequential setting), we call this the *inreach* of n, written $\mathsf{inr}(n)$ (intuitively, this is the set of keys k that can start at n and, following edges having k in their edgeset, reach the node containing k in its keyset). Figure 6.8 shows the inreach of each node in its top-right corner; the inreach of y_2

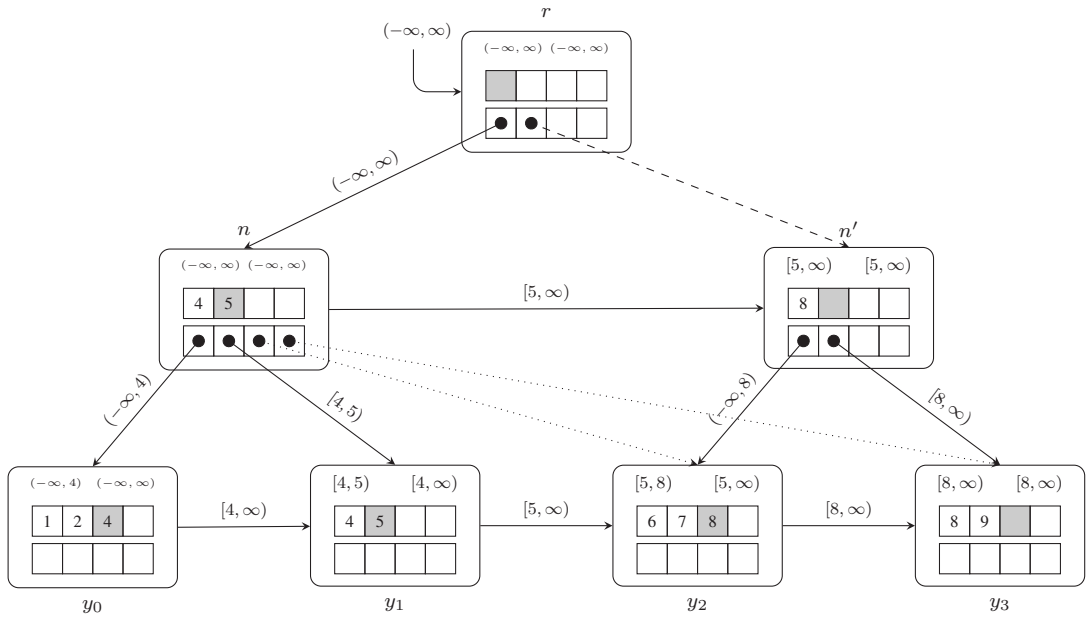

Figure 6.8: An example B-link tree in the middle of a split. Node n was full, and has been half-split and children y_2 and y_3 have been transferred to the new node n' (old edges are shown with dotted lines), but the complete-split has yet to add n' to the parent r (the dashed edge). Each node now additionally contains its inset (see §5.4) in the top left and the inreach (defined in this section) in the top right. The label with a curved arrow to the top-left of the root is its inflow (explained in §5.4).

is $[5,\infty)$ despite its inset's being only $[5,8)$ because it can still reach nodes with keys in $[8,\infty)$ in their keyset via link edges.

We define the inreach to be the solution to the following fixpoint equation

$$\forall n \in N.\ \mathsf{inr}(n) = \mathit{in}(n) \cup \bigcup_{n' \in N} \mathsf{es}(n', n) \cap \mathsf{inr}(n')$$

where in is any inflow such that $\mathit{in}(r) = \mathsf{KS}$. This may look identical to the definition of inset, but there is a subtle, but vital, difference: by not constraining the inflow of non-root nodes, we enable the split operation to add flow to nodes it has split to ensure that their inreach records the fact that they can still reach keys k that were moved to other nodes. For example, in Figure 6.9 when the full-split adds the edge (r, n') and re-routes keys in $[5,\infty)$ to take (r, n') instead of (r, n), then n's inset is reduced from $(-\infty,\infty)$ to $(-\infty,5)$.

Our solution to this is to *increase the inflow* of n to make up for the removed keys. We will formally justify this operation in §6.2.3, but at a high-level what we do is increase $\mathit{in}(n)$ from $\emptyset$ to $[5,\infty)$, thereby preserving its inreach of $(-\infty,\infty)$. Intuitively, this operation does not violate any

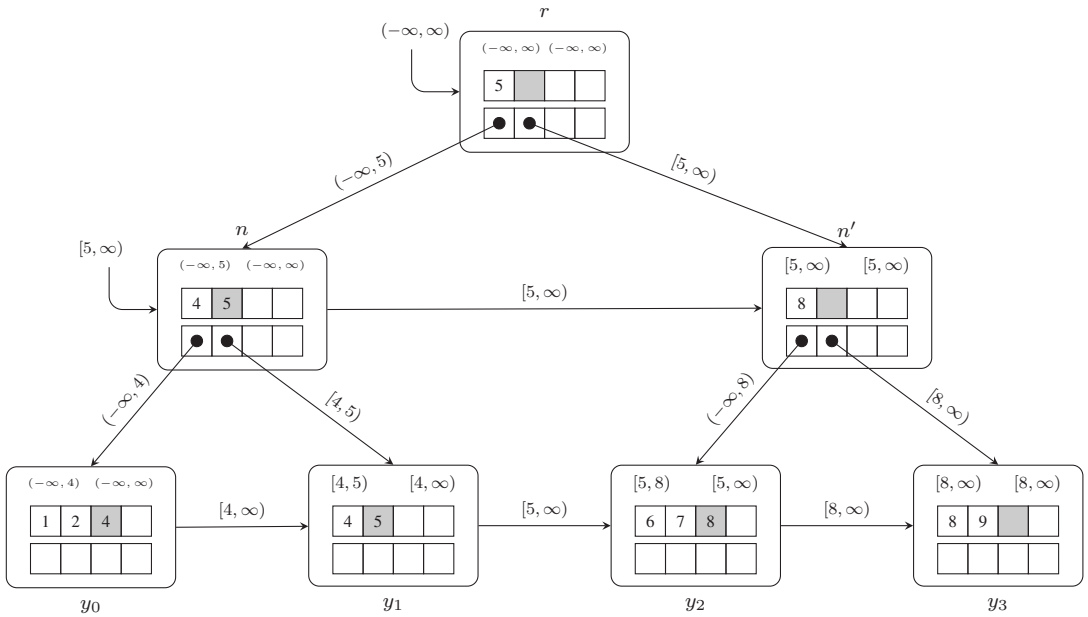

Figure 6.9: The B-link tree from Figure 6.8 after the full-split of n has completed. Note that while n's inset has been reduced to $(-\infty, 5)$, its inreach is preserved at $(-\infty, \infty)$.

other invariant because the newly added keys $[5, \infty)$ are propagated via the link edge to a node that has those keys in its inset (n'). Thus, this increase in inflow does not change the keyset of any node.

We have one final issue to solve: as it stands, the full-split does not preserve the interface of the region $\{r, n, n'\}$ because the outflow to y_2 and y_3 has increased. For example, the outflow from n' to y_2 was $[5, 8)$ before the full-split (the edge label in Figure 6.8 is the edgeset, which is $(-\infty, 8)$, but since the inset of n' is $[5, \infty)$, keys below 5 do not arrive at n' to be part of its inflow). However, after the full-split, the outflow from n' to y_2 is $\{[5, 8), [5, 8)\}$, i.e. the multiset where all keys in $[5, 8)$ have multiplicity 2. This is because one copy of keys in $[5, 8)$ arrive from the newly introduced inflow on n, and one copy from r.

The problem is in our encoding of edgesets and inreach in the flow framework: we had to use multisets instead of sets because sets are not a flow domain. Our solution is to tweak the edge functions from $e_{\mathsf{es}(n,n')} := (\lambda X.\ \mathsf{es}(n, n') \cap X)$ to $e_{\mathsf{es}(n,n')} := (\lambda X.\ \{k \rightarrowtail (k \in \mathsf{es}(n, n') \cap X\ ?\ 1 : 0)\})$, essentially projecting the multiset intersection back to a set, and preventing multiple copies of keys from being propagated.

We now have an invariant for `traverse`: $k \in \mathsf{inr}(n)$. This is true at the root, because $\mathsf{KS} = in(r) \subseteq \mathsf{inr}(r)$, and it is preserved during traversal since `findNext` follows edges with k in the edgeset. We will ensure that no concurrent operations reduce the inreach of any node by adding an appropriate constraint to the search structure predicate CSS in §6.2.2.

$$\mathsf{inr}(I_n, n) \coloneqq I_n.in(n) \qquad \mathsf{outs}(I_n) \coloneqq \bigcup_{n' \notin \mathsf{dom}(I_n)} \mathsf{outs}(I_n, n')$$
$$\mathsf{outs}(I_n, n') \coloneqq I_n.out(n') \qquad \mathsf{ks}(I_n, n) \coloneqq \mathsf{inr}(I_n, n) \setminus \mathsf{outs}(I_n, n)$$

Figure 6.10: The definition of keyset and associated quantities for the link template proof.

$$\mathsf{inFP}(n) \coloneqq \boxed{\circ\,\{n\}}^{\gamma_f}$$

$$\mathsf{inInr}(k, n) \coloneqq \exists R.\, \boxed{\circ R}^{\gamma_{i(n)}} * k \in R$$

$$\mathsf{N}(n, I_n, C_n) \coloneqq \mathsf{node}(n, I_n, C_n) * \boxed{\circ(\mathsf{ks}(I_n, n), C_n)}^{\gamma_k} * \mathsf{dom}(I_n) = \{n\} * \boxed{\tfrac{1}{2}\, I_n}^{\gamma_{h(n)}} * \mathsf{inFP}(n)$$

$$\varphi(r, I) \coloneqq \overline{\mathcal{V}}(I) \wedge I.in(r) = \mathsf{KS} \wedge I.out = \lambda_0$$

$$\mathsf{CSS}(r, C) \coloneqq \exists I.\, \boxed{\bullet I}^{\gamma_I} * \varphi(r, I) * \boxed{\bullet(\mathsf{KS}, C)}^{\gamma_k} * \boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f} * \mathop{\Large *}_{n \in \mathsf{dom}(I)} \Big(\exists b.\, \mathsf{lk}(n) \mapsto b * (b\ ?\ \mathsf{True} : \exists C_n.\, \mathsf{N}(n, I_n, C_n)) * \boxed{\circ I_n}^{\gamma_I} * \boxed{\bullet \mathsf{inr}(I_n, n)}^{\gamma_{i(n)}} * \boxed{\tfrac{1}{2}\, I_n}^{\gamma_{h(n)}} * \mathsf{dom}(I_n) = \{n\} \Big)$$

Figure 6.11: The definition of CSS used by the link template proof.

We amend the definition of keyset used in the give-up proof accordingly (Figure 6.10). The inreach is defined to be the flow of each node, the definition of outset is unchanged and the keyset of each node n is defined to be $\mathsf{inr}(n) \setminus \mathsf{outs}(n)$. This means that when `findNext` returns `None`, $k \in \mathsf{inr}(n)$ by the traversal invariant and $k \notin \mathsf{outs}(n)$ by the specification of `findNext`. Thus $k \in \mathsf{ks}(n)$, which by KS-UPD is sufficient to ensure correctness of the decisive operation (the proof is the same as the one we performed for the give-up template).

6.2.2 PROOF OF THE LINK TEMPLATE

We need to make a few more changes to the give-up proof in order to verify the link template. In particular, we need to add a few more types of ghost state to capture some of the special behaviour of the link template.

$\{\mathsf{node}(n, I_n, C_n) * k \in \mathsf{inr}(I_n, n)\}$
`findNext n k`
$\left\{v.\ \mathsf{node}(n, I_n, C_n) * \left(v = \mathsf{None} * k \notin \mathsf{outs}(I_n) \right.\right.$
$\left.\left. \vee\ v = \mathsf{Some}(n') * k \in \mathsf{outs}(I_n, n')\right)\right\}$

$\{\mathsf{node}(n, I_n, C_n) * k \in \mathsf{ks}(I_n, n)\}$
`decisiveOp` ω `n k`
$\left\{v.\ \mathsf{node}(n, I_n, C_n') * \left(v = \mathsf{None} * C_n = C_n' \right.\right.$
$\left.\left. \vee\ v = \mathsf{Some}(v') * \Psi_\omega(k, C_n, C_n', v')\right)\right\}$

$\mathsf{node}(n, I_n, C_n) * \mathsf{node}(n, I_n', C_n') \mathrel{-\!\!*} \mathsf{False}$

Figure 6.12: Assumptions the link template proof makes on helper functions and implementation-specific predicates. These are defined and proved by implementations.

Figure 6.11 contains our definition of the search structure predicate CSS and related predicates that we will use for the link template proof. As with the give-up proof, we use the authoritative RA of flow interfaces at location γ_I for the flow-based reasoning, the keyset RA from §4.5 to lift local updates to global ones, the authoritative RA of sets of nodes (and the accompanying inFP predicate) to encode the domain of the search structure. But we add two new types of ghost state to this definition.

First, we use fractional RAs at locations $\gamma_{h(n)}$ for each node n to store one half of the node's singleton interface I_n inside N and the other half inside CSS. Since fractional RAs can be updated only when both halves are together, this prohibits other threads from modifying the interface of n when one thread has locked n and removed $\mathsf{N}(n, I_n, C_n)$ from CSS.

Second, we use an authoritative RA of sets of keys, at locations $\gamma_{i(n)}$ for each node n, to encode the inreach of each node. This RA has similar rules as the authoritative RA of sets of nodes at location γ_f (i.e. AUTH-SET-UPD, AUTH-SET-SNAP, and AUTH-SET-UPD). In our proofs, we use these rules so that threads can take snapshots of a node n's inreach and assert that a given key is in it even when they have not locked n. We introduce a new predicate $\mathsf{inInr}(k, n)$ (defined in Figure 6.11) to represent the knowledge a thread has about n's inreach after taking a snapshot as described above. If a thread owns $\mathsf{inInr}(k, n)$, then it knows that k is in the inreach on n. This knowledge is stable under interference of other threads because in the link template, threads only increase the inreach of a node.

Before we describe the link template proof, we present the assumptions it makes about its implementation (summarized in Figure 6.12). Our specifications say that `findNext` is given a node n satisfying $\mathsf{node}(n, I_n, C_n)$ and returns None if k is not in the outset of n else $\mathsf{Some}(n')$ such that k is in the outflow to n' (by our definition of edge functions, this means $k \in \mathsf{es}(n, n')$). Similarly,

decisiveOp expects a node $\mathsf{node}(n, I_n, C_n)$ such that k is in the keyset of n. If decisiveOp returns None then it returns the node unchanged. On the other hand, if it returns $\mathsf{Some}(v')$ then the node is now $\mathsf{node}(n, I_n, C'_n)$, and the return value satisfies the search structure specification with respect to the old and new contents of the node n ($\Psi_\omega(k, C_n, C'_n, v')$). Finally, we assume that the heap representation predicate $\mathsf{node}(n, I_n, C_n)$ implies that we have ownership of the heap location n; in particular, we need the property that it cannot be duplicated, hence owning two copies of it implies False.

We now turn to the template proof, shown in Figure 6.13. Recall that our objective is to prove the atomic triple for cssOp from Figure 6.1, using the helper function specifications listed in Figure 6.12 and the lock module specification from Figure 6.2.

The specification of traverse that we prove is almost the same as in the give-up proof, with the addition of $\mathsf{inInr}(k, n)$:

$$\mathsf{inFP}(n) * \mathsf{inInr}(k, n) \mathrel{-\!\!*} \\ \langle\, C.\ \mathsf{CSS}(r, C) \,\rangle\ \texttt{traverse r n k}\ \langle\, v.\ \mathsf{CSS}(r, C) * \mathsf{N}(v, I_v, C_v) * k \in \mathsf{ks}(I_v, v) \,\rangle$$

This specification says that if n is in the footprint of the search structure, and k is in the inreach of n, then traverse operates on the entire search structure and returns a locked node v such that k is in the keyset of v, while leaving the global contents unmodified.

As before, we start the proof by applying the high-level specification of lockNode from Figure 6.2. This adds $\mathsf{N}(n, I_n, C_n)$ to the thread-local state. Before we can move on to the call to findNext, note that findNext's precondition requires $k \in \mathsf{inr}(I_n, n)$ but we have $\mathsf{inInr}(k, n)$. The difference is subtle, but luckily we can convert one to the other using the following lemma that can be proved from the definitions of the involved predicates and AUTH-SET-VALID:

ININR-INR
$$\mathsf{CSS}(r, C) * \mathsf{N}(n, I_n, C_n) * \mathsf{inInr}(k, n) \mathrel{-\!\!*} k \in \mathsf{inr}(I_n, n)$$

Since we need the CSS to apply ININR-INR, we open the precondition and apply this rule. This is a purely logical step that doesn't modify the shared state, so we can close the precondition again (i.e. we use AU-ABORT) and get the state shown in line 6. We now have all the resources needed to apply findNext's specification (Figure 6.12).

As before, if findNext succeeds, we directly obtain the postcondition of traverse, so we commit (i.e. this is the linearization point) and complete this branch of the proof. If it fails, then as before we need to prove the precondition of findNext for n' for the recursive call. This uses similar reasoning to the give-up proof, but here we must additionally establish $\mathsf{inInr}(k, n')$. We do this using the following lemma that says that if k is in the inreach of n and is in the outset from n to n', then it must be in the inreach of n':

INREACH-STEP
$$\mathsf{CSS}(r, C) * \mathsf{N}(n, I_n, C_n) * k \in \mathsf{inr}(I_n, n) * k \in \mathsf{outs}(I_n, n') \mathrel{-\!\!*} k \in \mathsf{inr}(k, n')$$

This lemma can be proved using the definition of inreach and some lemmas of the flow framework encoding that we described in §6.2.1. Again, we open the precondition to apply this lemma, closing

```
1  inFP(n) * inInr(k,n) -*⟨ C. CSS(r,C) ⟩
2  let rec traverse n k =
3    {inFP(n) * inInr(k,n)}
4    lockNode n;
5    {inFP(n) * inInr(k,n) * N(n,I_n,C_n)}
6    {N(n,I_n,C_n) * k ∈ inr(I_n,n)}
7    match findNext n k with
8    | None -> {N(n,I_n,C_n) * k ∈ ks(I_n,n)}
9      n
10   | Some n' -> {N(n,I_n,C_n) * k ∈ inr(I_n,n) * k ∈ outs(I_n,n')}
11     {N(n,I_n,C_n) * inFP(n') * inInr(k,n')}
12     unlockNode n; {inFP(n') * inInr(k,n')}
13     traverse n' k
14 ⟨ v. CSS(r,C) * N(v,I_v,C_v) * k ∈ ks(I_v,v) ⟩
15
16 ⟨ C. CSS(r,C) ⟩
17 let rec cssOp ω r k =
18   {inFP(r) * inInr(k,r)}
19   let n = traverse r k in
20   {N(n,I_n,C_n) * k ∈ ks(I_n,k)}
21   match decisiveOp ω n k with
22   | None -> {N(n,I_n,C_n)}
23     unlockNode n; {True}
24     cssOp ω r k
25   | Some res ->
26     {node(n,I_n,C'_n) * Ψ_ω(k,C_n,C'_n,res) * k ∈ ks(I_n,n) * ···}
27     (* Linearization point: open precondition *)
28     ⟨ ◦(ks(I_n,n),C_n)^γ_k * Ψ_ω(k,C_n,C'_n,res) * k ∈ ks(I_n,n) * •(KS,C)^γ_k * ··· ⟩
29     ⟨ ◦(ks(I_n,n),C'_n)^γ_k * Ψ_ω(k,C,C',res) * •(KS,C')^γ_k * ··· ⟩ (* By KS-UPD *)
30     ⟨ N(n,I_n,C'_n) * CSS(r,C') * Ψ_ω(k,C,C',res) ⟩
31     unlockNode n;
32     ⟨ CSS(r,C') * Ψ_ω(k,C,C',res) ⟩ (* Prove postcondition, commit *)
33   {True}
34   res
35 ⟨ v. CSS(r,C') * Ψ_ω(k,C,C',v) ⟩
```

Figure 6.13: The link template algorithm with a proof outline.

it again without modifying it. We can then use the high-level specification of `unlockNode` to return the node n, and obtain the postcondition of `traverse` by using induction on the recusrive call.

The `cssOp` operation begins with a call to `traverse` on line 19. To satisfy `traverse`'s precondition, we need to open the precondition and take a snapshot of the global footprint (using AUTH-SET-SNAP and $\varphi(r, I) \Rightarrow r \in \mathsf{dom}(I)$), obtaining $\mathsf{inFP}(r)$. Also, $\varphi(r, I) \Rightarrow k \in \mathsf{inr}(I_r, r)$ so we also take a snapshot of r's inreach at ghost location $\gamma_{i(r)}$ to add $\mathsf{inInr}(k, r)$ to our context. The resulting context is depicted in line 18.

To call `traverse` we also need $\mathsf{CSS}(r, C)$, so we need to open the precondition again. This is allowed because since `traverse` has an atomic triple, it behaves atomically and we can open atomic preconditions (i.e. use AU-ABORT and AU-COMMIT) around calls to it. After `traverse` returns, we add its postcondition in line 14 to our context (minus $\mathsf{CSS}(r, C)$, which needs to be given back to re-establish `cssOp`'s precondition since we do not commit here). The next step is the call to `decisiveOp`, for which we already have the precondition in our context.

We then look at the two possible outcomes of `decisiveOp`. In the case where it returns `None`, our context is unchanged, so we execute `unlockNode` using the $\mathsf{N}(n, I_n, C_n)$ in our context. We can use the specification of `cssOp` on the recursive call on line 24 to complete this branch of the proof.

On the other hand, if `decisiveOp` succeeds, we get back a modified node $\mathsf{node}(n, I_n, C'_n)$ with new contents C'_n that satisfies the search structure specification $\Psi_\omega(k, C_n, C'_n, \mathsf{res})$ locally (line 26). We now need to show that this modification results in `cssOp`'s postcondition; this is essentially the *linearization point* of this algorithm.

To do this, we again open the atomic precondition $\mathsf{CSS}(r, C)$. We now have the context in line 28 (we have also expanded $\mathsf{N}(n, I_n, C'_n)$), and now we can apply our ghost update KS-UPD to update the global contents and get the context in line 29. In particular, we have $\Psi_\omega(k, C, C', \mathsf{res})$ and $\mathsf{CSS}(r, C')$, which allows us to "commit" and establish the postcondition. We finally apply the specification of `unlockNode` using the remaining $\mathsf{N}(n, I_n, C'_n)$, and complete the proof.

6.2.3 MAINTENANCE OPERATIONS.

As with the give-up, maintenance operations for the link technique need to be proved separately. The specification for a maintenance operation `maintOp` is the same:

$$\langle\, C.\ \mathsf{CSS}(r, C) \,\rangle\ \texttt{maintOp r}\ \langle\, \mathsf{CSS}(r, C) \,\rangle$$

Again, this triple gives the maintenance operation the permission to modify the search structure $\mathsf{CSS}(r, C)$ as long as it instantaneously modifies the structure to another valid search structure $\mathsf{CSS}(r, C)$ with the same contents. While we once again omit the full proofs here, we describe them at a high-level below and show the new lemmas needed.

For the B-link tree, the maintenance operations are the half-split, full-split, and merge. The interesting part of their proofs is in showing that they do not decrease the inreach of any node, which for the half-split and merge is easy to do. The half-split also requires us to reason about the creation

of a new node, which we do using the domain extension notion (Definition 6.1) and frame-preserving update from §6.1.2.

For the full-split, we have discussed how we need a frame-preserving update that allows increasing the inflow of nodes that allows us to show that the full-split preserves the inreach of all modified nodes. This is formally justified by a notion of interface extension that allows increasing the inflow of the modified region. Note that this definition is only for flow domains that are positive monoids (Chapter 3), which is true of the flow domain that we use, multisets of keys.

Definition 6.2 Given a positive flow domain, a flow interface I' is an *inflow extension* of interface I, written $I \preccurlyeq I'$, if and only if

1. $\mathsf{dom}(I) = \mathsf{dom}(I')$,
2. $\forall n \in I.\ I.in(n) \leq I'.in(n)$, and
3. $I.out \equiv I'.out$.

This definition allows I' to differ from I by having a larger inflow, as long as the domains and the outflow of the of the modified region are exactly the same. Similarly, we can show that replacing an interface with an inflow extension is a frame-preserving update in the flow interface RA:

$$\textsc{flowint-inf-upd}\quad \frac{I_1 \preccurlyeq I_1' \qquad \text{the flow domain is positive}}{(\bullet I_1 \oplus I_2, \circ I_1) \rightsquigarrow \{(\bullet I_1' \oplus I_2, \circ I_1') \mid (I_1 \oplus I_2) \preccurlyeq (I_1' \oplus I_2)\}}$$

We use this lemma in the full-split to update the flow interface ghost state in such a way as to preserve the inreach of all modified nodes.

6.3 THE LOCK-COUPLING TEMPLATE

The lock-coupling template (Figure 6.14) uses the hand-over-hand locking scheme to ensure that no thread is negatively interfered with while it is on its traversal. Unlike the other two templates in this chapter, every thread always holds at least one lock while traversing from one node to the next. This means that no other thread can overtake this thread, or perform any modification that would invalidate this thread's search.

This template can be instantiated to an implementation that uses a sorted singly-linked list, where each node contains a single key. Insert operations create a new node and link the node into the appropriate position of the list, while delete operations unlink the node containing the operation key from the list by swinging the pointer from the predecessor node to the successor node (as shown in Figure 5.1).

The lock-coupling technique is a simpler and less efficient concurrency technique. It can be proved, on paper at least, using the standard conflict-preserving serializability technique [Bernstein

```
let rec traverse p n k =
  match findNext n k with
  | None -> (p, n)
  | Some n' ->
      unlockNode p;
      lockNode n;
      traverse n n' k
```

```
let rec searchStrOp ω r k =
  lockNode r;
  match findNext r k with
  | None -> searchStrOp ω r k
  | Some n ->
      let (p, n) = traverse r n k in
      let res = decisiveOp ω p n k in
      unlockNode p;
      unlockNode n;
      res
```

Figure 6.14: The lock-coupling template algorithm. Like all our templates, the main method is `cssOp` and `traverse` is an auxiliary method used for recrusive traversal.

et al., 1987]. Thus, we omit the details of its proof using our technique here. The full proof is available online in our public repository of machine-checked proofs (see §6.5), along with the other proofs in our book.

6.4 VERIFYING IMPLEMENTATIONS

To obtain a verified implementation of one of the templates, one needs to specify the concrete representation of a node by defining the `node` predicate and provide code for the helper functions that satisfies the specifications assumed by the template in question. For example, to verify an implementation of the give-up template, one needs to provide implementations for `node`, `inRange`, `findNext`, and `decisiveOp` that satisfy the assumptions listed in Figure 6.5.

We would like to re-emphasize that these specifications use sequential Hoare triples and assert ownership of only locked nodes. Thus, if their implementations are sequential code, we can verify them using an off-the-shelf separation logic tool that can verify sequential heap-manipulating code. Such tools typically provide better automation, and can help speed up the verification process.

In the next section we will discuss the tools we used and the implementations we verified for each of the three template algorithms seen in this chapter.

6.5 PROOF MECHANIZATION AND AUTOMATION

This section evaluates the techniques presented so far by mechanically verifying the three template algorithms from this chapter and some real-world implementations based on them. Our case studies are summarized in Figure 6.15.

We verify the give-up, link, and lock-coupling template proofs that were described above. These proofs have been mechanically checked using the Coq proof assistant, building on the formalization of Iris [Jung et al., 2015]. The proofs parameterize over the implementation of the helper functions (e.g. `decisiveOp`, `findNext`, etc.) and the heap representation predicate `node`.

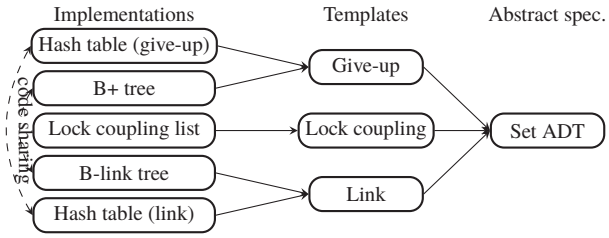

Figure 6.15: The templates and implementations in our case studies. The arrows indicate that all three templates satisfy the abstract specification of a Set ADT (as defined in Figure 6.1), and link implementations to the template that they instantiate.

For the link and give-up templates, we have derived and verified implementations based on B-trees and hash tables. In particular, we verify the B-link tree implementation described in Chapter 2. For the lock-coupling template we have considered a sorted linked list implementation.

These concrete implementations have been verified using the separation logic based deductive program verifier GRASShopper [Piskac et al., 2014]. As the tool uses SMT solvers to largely automate the verification process, this provided us with a substantial decrease in effort. While we do not have, as of now, a formal proof for the transfer of proofs between Iris and GRASShopper, note that Iris is expressive enough to support all the reasoning that we do in GRASShopper, but comes with significant additional manual effort.

Our verification effort includes a mechanization of the meta-theory of flows [Krishna et al., 2020b] (i.e. that flow interfaces form an RA). Our formalization is parametric in the flow domain (i.e. the underlying cancellative, commutative monoid). We also provide instantiation of the meta-theory for the specific flow domains used in our proofs (e.g. multisets). We have duplicated this effort in Iris/Coq and GRASShopper in order to make the two parts of our verification self-contained. The formalization is available as two standalone libraries that can be reused for other flow-based proofs in these systems.

In addition to the helper functions of each data structure that are assumed by the templates, we have also verified the split operations for B-link trees. The B-link tree uses a two-part split operation: a half-split that creates a new node, transfers half the contents from a full node to this new node, and adds a link edge; and a full-split that completes the split by linking the original node's parent to the new node. For the split operations, we assume a harness template for a maintenance thread that traverses the data structure graph to identify nodes that are amenable to half splits. While we have not verified this harness, we note that it is a variation of our lock-coupling template where the abstract specification leaves the contents of the data structure unchanged. For the implementations of half and full splits, we verify that the operation preserves the flow interface of the modified region as well as its contents.

The full development of our mechanization effort is available online at
`https://github.com/nyu-acsys/template-proofs/tree/css_book`.

Table 6.1: Summary of templates and instantiations verified in Iris/Coq and GRASShopper. For each algorithm or library, we show the number of lines of code, lines of proof annotation (including specification), total number of lines, and the proof-checking / verification time in seconds.

Module	Code	Proof	Total	Time
Templates (Iris/Coq)				
Flow library	0	2803	2803	114
Link template	14	487	501	55
Give-up template	18	390	408	49
Lock-coupling template	26	980	1006	238
Total	**58**	**4660**	**4718**	**456**
Implementations (GRASShopper)				
Flow library	0	721	721	9
Array library	143	320	463	9
B+ tree	63	99	162	21
B-link (core)	85	161	246	36
B-link (half split)	34	192	226	94
B-link (full split)	17	137	154	697
Hash table (link)	54	99	153	10
Hash table (give-up)	60	138	198	13
Lock-coupling list	59	300	359	51
Total	**515**	**2167**	**2682**	**940**

Table 6.1 provides a summary of our development. Experiments have been conducted on a laptop with an Intel Core i7-5600U CPU and 16GB RAM. We split the table into one part for the templates (proved in Coq) and one part for the implementations (proved in GRASShopper). We note that for the B-link tree, B+ tree and hash table implementations, most of the work is done by the array library, which is shared between all these data structures. The size of the proof for the lock-coupling list and maintenance operations is relatively large. The reason is that these involve the calculation of a new flow interface for the region obtained after the modification. This requires the expansion of the definitions of functions related to flow interfaces, which are deeply nested quantified formulas. GRASShopper enforces strict rules that limit quantifier instantiation so as to remain within certain decidable logics [Bansal et al., 2015, Piskac et al., 2013]. Most of the proof in this case involves auxiliary assertions that manually unfold definitions. The size of the proof could be significantly reduced with a few improved tactics for quantifier expansion [Leino and Pit-Claudel, 2016].

It is difficult to assess the overall time effort spent on verifying the link template algorithm, which was the first algorithm that we considered. The reason is that we designed our verification methodology as we verified the template. However, with all the machinery now in place, our ex-

perience is that verifying a new template algorithm is a matter of a few hours of proof effort. In fact, adapting the link template proof to the give-up template was straightforward and required only minor changes. Our experience with adapting implementation proofs is similar.

We believe that our case studies are representative of real-world applications and that our methodology can be widely applied. The template algorithms that we have verified focus on lock-based techniques with fixed linearization points inside a decisive operation. In fact, many real-world applications perform better using lock-based algorithms instead of lock-free algorithms as the latter tend to copy data more[3]. On the other hand, our methodology does not require locking, and can be extended to prove lock-free algorithms such as the Bw-tree [Levandoski and Sengupta, 2013] (The Bw-tree is an example of a multi-copy search structure, which we examine in Chapter 7). While our methodology can, in theory, be applied to any search structure implementation, there are implementations that use very specific concurrency techniques that cannot be used by other heap representations (e.g. Harris' list [Harris, 2001]; see §10.2 for a discussion of the Harris list). Our technique would give us a "single-use" template in such cases, but this would still structure the proof and make it simpler to construct and verify.

[3]For instance, Apache's CouchDB uses a B+ tree with a global write lock; BerkeleyDB, which has hosted Google's account information, uses a B+ tree with page-level locks in order to trade-off concurrency for better recovery; and java.util.concurrent's hash tables lock the entire list in a bucket during writes, which is more coarse-grained than the one we verify.

CHAPTER 7

Verifying Multicopy Structures

7.1 OVERVIEW

In Chapter 6 we demonstrated how to simplify the verification of concurrent search structures by abstracting the common concurrency algorithms underlying diverse implementations such as B-trees and hash tables into templates that can be verified once and for all. The template algorithms we have considered so far handle only search structures that perform all operations on keys *in place*. That is, an operation on key k searches for the unique node containing k in the structure and then performs any necessary updates on that node. Since every key occurs at most once in the data structure at any given moment, we refer to these structures as *single-copy structures*.

Single-copy structures demonstrate good read performance. However, some applications, such as event logging, favor write performance over read performance, possibly at the cost of increased memory overhead. This demand is met by data structures that store updates to a key k *out of place* at a new node instead of overwriting a previous copy of k that was already present in some other node. Performing out-of-place updates can improve write performance because the updates can be done in constant time (e.g., always at the head of a list). A consequence of this design is that the same key k can now be present multiple times simultaneously in the data structure. Hence, we refer to these structures as *multicopy structures*.

Multicopy structures are commonly used in scenarios where the search structure is spread over multiple media such as memory, solid-state drives, and hard disk drives. Each medium necessitates a different data structure to make use of its particular characteristics. We thus have a data structure composed of data structures. In this chapter, we treat the inner data structures as nodes, and study template algorithms for the outer data structure. Inner data structures are single-copy search structures, and so the algorithms and proof techniques studied in the previous chapters can be applied to them.

Examples of multicopy structures include the differential file structure [Severance and Lohman, 1976], log-structured merge (LSM) tree [O'Neil et al., 1996], and the Bw-tree [Levandoski et al., 2013]. The differential file structure and LSM tree, in particular, can be tuned by implementing workload- and hardware-specific data structures at the node level. In addition, research has been directed towards optimizing the layout of nodes and developing different strategies for the maintenance operations used to reorganize these data structures. This has resulted in a variety of implementations today (e.g. [Dayan and Idreos, 2018, Raju et al., 2017, Thonangi and Yang, 2017, Wu et al., 2015]. For a comprehensive review of the literature on LSM trees, see [Luo and Carey, 2020]).

Despite the differences between these implementations, they generally follow the same high-level algorithms for the core search structure operations.

In the following chapters, we derive template algorithms for such concurrent multicopy structures from high-level descriptions of their operations. These templates can then be instantiated to the differential file and LSM tree implementations. We further show how the flow framework can be used to prove their correctness. A new technical challenge that we need to address here is that the multicopy design complicates the linearizability proof: in a multicopy structure, the logical value of a key k is defined as the value associated with the latest copy of k.

However, a thread T searching for the value associated with k may be unaware of updates to that value that have happened after T started its search. As a consequence, T may return a value that is not associated with the latest copy of k at the time of return. This by itself does not violate correctness as long as the copy of k that T operated on was the latest copy at some point in time after T started its search - a property we refer to as *search recency*.

The correctness argument is now more involved, however, because T's linearization point is no longer a fixed step in its decisive operation (as was the case for the templates we have considered so far) but can be some point during the preceding data structure traversal. In general, the linearization point thus depends on the history of the computation and the interference of other threads. We propose an appropriate flow domain and flow to track such history information in a local proof of the search recency property.

In the remainder of this chapter, we will describe the differential file structure [Severance and Lohman, 1976] and LSM tree [O'Neil et al., 1996] in more detail. We will then derive an abstract notion of multicopy structures similar to the abstract single-copy structures in the edgeset framework. We further provide general atomic specifications for the operations on multicopy structures and show that any concurrent multicopy template algorithm that obeys these specification is linearizable. In chapters 8 and 9 we then discuss two template algorithms that generalize differential file structures and LSM trees, respectively, and prove that they satisfy the desired specifications.

7.2 DIFFERENTIAL FILE STRUCTURES

The differential file structure is one of the earliest examples of a key-value store that allows out-of-place updates. The structure stores the data in a *main file* on disk. However, when key-value pairs are inserted or deleted, the changes are first recorded in a *differential file* in memory instead of modifying the main file directly. In the following, we will refer to the differential file as the *root node* and the main file as the *disk node* of the data structure. Figure 7.1 shows a potential state of a differential file structure with root node r and disk node n.

The insert and delete operations are performed by a single generic operation referred to as an *upsert*. The upsert operation takes a key-value pair (k, v) and inserts it into the root node. If an entry (k, v') already exists in that node, then v' is replaced by v. Otherwise, the new pair is inserted. To delete a key k from the structure, one upserts the pair $(k, \square)$ where $\square$ is a dedicated *tombstone* value used to indicate that k has been deleted.

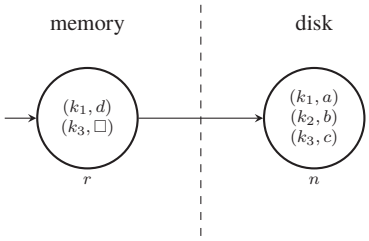

Figure 7.1: A differential file structure

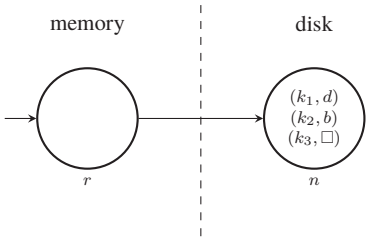

Figure 7.2: A differential file structure obtained after reorganizing data in Fig 7.1

When searching for the value associated with a given query key k, the search thread consults the root node first. If k is not found, then the thread proceeds to the disk node. For instance, a search for key k_2 on the differential file structure in Fig. 7.1 will return b, while a search for k_3 will return $\square$ indicating that k_3 has been deleted.

Periodically (e.g., after a certain time period or when the root node becomes full), a maintenance operation reorganizes the structure by moving the contents of the root node to the disk node, leaving the root node empty so it can begin collecting changes once again. In case of a conflict when moving the data, i.e. if a copy of key k exists in both nodes, then the copy in the root node is kept as it is more recent. Figure 7.2 shows the structure obtained after reorganizing the structure shown in Fig. 7.1. Here, the copy $(k_3, \square)$ in the root node is kept over (k_3, c) in the disk node.

Each of the two nodes is itself implemented as a single-copy structure that may itself consist of many (sub)nodes. How these node-level data structures are implemented depends on the underlying storage medium and the sequential or concurrent algorithm used. If a concurrent algorithm is used, it could be verified using the single-copy templates from previous chapters.

In the case of a differential file structure, all updates, inserts, and deletes happen at the root. The disk node is read-only except when maintenance operations take place during which time it is exclusively locked and is updated using a sequential process. In the remainder of our discussion, we will therefore treat node-level operations abstractly as atomic operations on the set of key-value pairs stored in a node and focus our attention on concurrency of the top-level structure, which in this case consists of the root and disk nodes, but which we will generalize in the next section.

7.3 LOG-STRUCTURED MERGE TREES

Modern LSM (Log-Structured Merge) tree implementations have evolved out of differential file structures. They target applications that require high write performance such as file systems for storing transactional log data. The LSM tree can be seen as a generalization of the differential file structure, in the sense that it replaces the single disk node n with a list of disk nodes $n_1, n_2, \ldots, n_l$. Fig. 7.3 shows an example. As in the case of differential file structures, the implementation of the node-level structures can depend on the storage medium. Most modern implementations use skip lists for memory nodes and sorted string tables of fixed size for the nodes stored on disk. [1] The size of the disk nodes typically increases exponentially as one moves down the list (i.e. further from the root).

The upsert and search operations essentially follow the same idea as in the differential file structure. The upsert operation takes place at the root node r. A search for a query key k traverses the list starting from the root node and retrieves the value associated with the first k that is encountered. If the retrieved value is $\square$ or if no entry for k has been found after traversing the entire list, then the search determines that k is not present in the data structure. Otherwise, it returns the retrieved value. For instance, a search for key k_1 on the LSM tree depicted in Fig. 7.3 would determine that this key is not present since the retrieved value is $\square$ from node n_1. Similarly, k_4 is not present since there is no entry for this key. On the other hand, a search for k_2 would return d and a search for k_3 would return c.

To prevent the root node from growing too large, the LSM tree performs *flushing*. As the name suggests, the flushing operation flushes the data from the root node to the disk by moving its contents to a newly created disk node. Figure 7.4 shows the LSM tree obtained from Fig. 7.3 after flushing the contents of r to the new disk node m.

To prevent the list from growing too long, the LSM tree performs *compaction*. The compaction operation merges the data from two (or more) nodes in the disk and replaces them with a single one. During the merge, if a key is present in more than one of the merged nodes, then the most recent (closer to the root) copy is kept, while stale copies are discarded. Figure 7.5 shows the LSM tree obtained from Fig. 7.3 after compacting nodes n_1 and n_2 to the new node m. Here, the copy of k_2 in n_2 has been discarded.

As is the case for differential file structures, the net effect of all these operations is that the logical value of key k is the most recently upserted value associated with key k.

7.4 MULTICOPY STRUCTURES

Analogous to the verification methodology described in Chapter 6, we abstract away from the data organization within the nodes, and treat the data structure as consisting of nodes in a mathematical directed acyclic graph. In current implementations and in our proof, this graph is a list, but the generalization to a directed graph is straightforward. Moreover, we will abstract from the concrete data

[1] RocksDB [Facebook, 2020], LevelDB [Google, 2020] and Apache HBase [Apache Software Foundation, 2020b] all use variants of concurrent skip list for in-memory data structure and SSTables for disk storage.

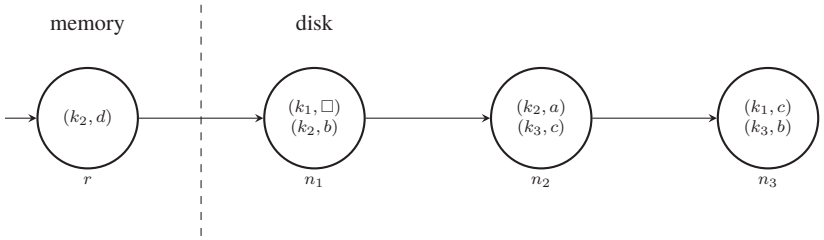

Figure 7.3: Top-level structure of an LSM tree

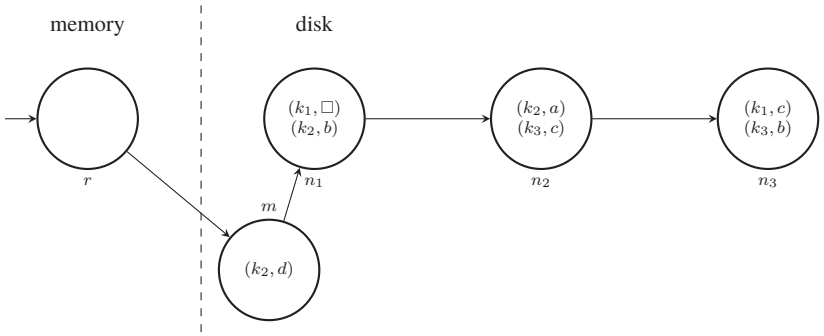

Figure 7.4: LSM tree obtained from Fig. 7.3 after flushing node r to disk

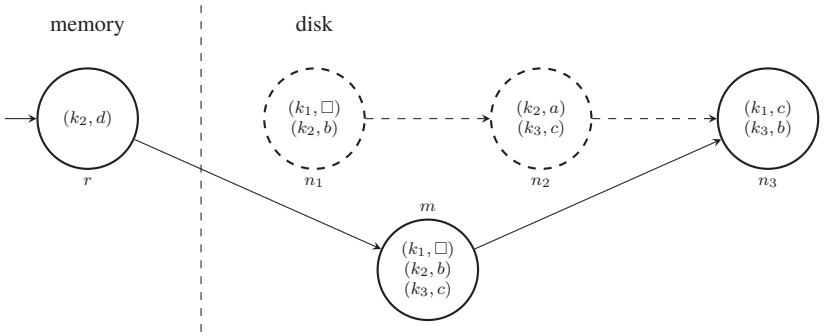

Figure 7.5: LSM tree obtained from Fig. 7.3 after compacting nodes n_1 and n_2

values associated with the keys in the data structure to keep the presentation simple. However, since multiple copies of any key k can be present in different nodes of the data structure simultaneously, we need a mechanism to differentiate between these copies. We therefore represent the entries of the contents of the data structure as pairs (k, t) where t is a timestamp uniquely identifying the point when this copy of k was upserted. We use a single global clock for this purpose. For example, $(k_3, 4)$ was upserted after $(k_2, 3)$, which was upserted after $(k_3, 1)$.

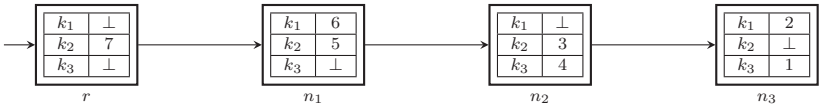

Figure 7.6: Abstract multicopy data structure graph for the LSM tree in Fig. 7.3.

Formally, let KS be the set of all keys. A multicopy structure is a directed acyclic graph $G = (N, E)$ with nodes N and edges $E \subseteq N \times N$. We assume that there is a dedicated *root node* $r \in N$ which uniquely identifies the structure (in case there are multiple instances that threads may operate on simultaneously). Each node n of the graph is labelled by its contents $C_n \colon \mathsf{KS} \to \mathbb{N}_\bot$, which is a map from keys to timestamps ($\mathbb{N}_\bot := \mathbb{N} \uplus \{\bot\}$). For a node n and its contents C_n, we say (k, t) is in the contents of n if $C_n(k) = t$. We denote the absence of key k in n by $C_n(k) = \bot$. As usual, for each edge $(n, n') \in E$ in the graph, the edgeset $\mathsf{es}(n, n')$ is the set of keys for which an operation arriving at a node n traverses (n, n'). We require that the edgesets of all outgoing edges of a node n are pairwise disjoint. Figure 7.6 shows a potential abstract multicopy structure graph consistent with the LSM tree depicted in Fig. 7.3. Here, all edges have edgeset KS. Observe that at this level of abstraction, we do not distinguish among insertions, updates, and deletions. Instead, we record the timestamp of the upsert for each copy of a key.

Sequential specification. As our goal is to prove linearizability of concurrent multicopy structure templates, we need a specification for a hypothetical sequential implementation that we take as the baseline of our verification. In principle, the desired sequential specification is that of a map ADT, i.e., the abstract contents of the data structure is a partial mathematical map M from keys to values. A `search` $r\, k$ returns the value $M(k)$ associated with the operation key k and an `upsert` $r\, k\, v$ updates M to $M[k \rightarrowtail v]$, associating k with the given value v. However, attempting to prove such a specification directly is difficult due to the fact that concurrent searches can actually return old values that are not consistent with the data structure's current abstract contents. We solve this problem by using the timestamp when an `upsert` operation takes effect in lieu of the value that is being upserted. That is, we view a multicopy structure abstractly as a map from keys to timestamps. In particular, when proving linearizability, this will allow us to reorder concurrent searches appropriately based on the timestamp that they return. This simplifies the correctness argument. One can easily recover a specification in terms of key/value maps by tracking the upserted values alongside the timestamps of the upsert. We omit this step from our discussion as it does not provide any substantial insight.

To formalize the above idea, we extend the ordering on natural numbers to $\mathbb{N}_\bot$ by defining $\bot < n$ for all $n \in \mathbb{N}$. The *logical (timestamp) contents* of a multicopy structure is the map $M \colon \mathsf{KS} \to \mathbb{N}_\bot$ that associates every key with its latest copy in the structure:

$$M(k) := \max \{t \mid \exists n \in N.\ C_n(k) = t\}$$

We call $M(k)$ the *logical timestamp* of key k. Note that $M(k) = \bot$ indicates that no copy of k exists in the structure, i.e. k has not been upserted so far.

Now suppose that $\overline{\mathsf{MCS}}(r, t, M)$ is a predicate that abstracts the state of a multicopy structure with root r by its logical contents M as well as the current value t of the global clock. We then require that `search` and `upsert` respect the following sequential specifications:

$$\forall t, M.\ \{\overline{\mathsf{MCS}}(r, t, M)\}\ \texttt{upsert r k}\ \{\overline{\mathsf{MCS}}(r, t+1, M[k \rightarrowtail t])\} \tag{7.1}$$

$$\forall t, M.\ \{\overline{\mathsf{MCS}}(r, t, M)\}\ \texttt{search r k}\ \{t'.\ \overline{\mathsf{MCS}}(r, t, M) * M(k) = t'\} \tag{7.2}$$

The sequential specification of `upsert` updates the logical timestamp of k to the current global clock value and increments the global clock. Similarly, the specification of `search` expresses that `search` returns the logical timestamp of its query key.

Unfortunately, we cannot just turn these sequential specifications into corresponding atomic triples that we can then use to verify concurrent template algorithms for multicopy structures. The problem is that, in contrast to the single-copy structures that we have considered so far, the linearization point of `search` is not fixed at the moment that a thread reaches its decisive operation, but depends on the upserts performed by concurrently executing threads. If we attempted to convert (7.2) directly into an atomic triple, this issue would be reflected in the proof of `search` in the form of non-fixed commit points of this atomic triple. While Iris provides mechanisms for reasoning about non-fixed commit points, the subset of the logic that we have been using in this book is ill-equipped to handle this additional proof complexity.

Rather than introducing advanced Iris features to deal with this situation, we derive alternative atomic specifications for the multicopy structure operations that require reasoning only about fixed commit points. However, this simplification comes at a small price: we have to relate the two types of specifications using a meta-level correctness argument that we prove only on paper.

Atomic specification: search recency. We start with an example that illustrates the issue of the dynamic linearization points of `search`. Consider a thread T_1 that starts a search for key k_3 in the multicopy structure depicted in Fig. 7.6. Since k_3 is not contained in the root node r, T_1 will proceed to the first disk node n_1. Suppose that at this point another thread T_2 concurrently upserts k_3 into the root node, say with timestamp 8. After T_2 completes its upsert, T_1 continues its search, proceeding to node n_2 where it finds an entry for k_3 and returns the associated timestamp 4 (and, in a full key-value implementation, the value associated with the upsert at timestamp 4). However, at this point we have $M(k_3) = 8$. Thus, this execution seemingly violates the postcondition of `search`'s sequential specification given above. Nevertheless, the execution is still linearizable. In particular, we can choose the start of T_1 as its linearization point since we have $M(k_3) = 4$ at this point, thus satisfying the postcondition of `search`. On the other hand, if we consider an alternative interleaving of the two thread's executions where thread T_2 completes its upsert before T_1 inspects the root node, but after T_1 starts its search, then T_1 will return 8 and its linearization point is the point when it reads the entry for k_3 in r. Thus, the choice of the linearization point of `search` depends on the relative speeds of concurrent upserts and searches.

More generally, we will see that `search(k)` is linearizable if it either returns the logical timestamp associated with k at the point when the search started, or any other copy of k that was upserted between the search's start time and the search's end time. We refer to this property as search recency.

A formal specification of `search` that captures the search recency property needs to relate the return value of `search` to the copies of the query key k that were concurrently upserted while the search executed. Our abstraction of a multicopy structure in terms of its logical contents M is not well-suited for this purpose because it loses too much information about the data structure's state: the logical contents tracks only the latest copies of keys but not stale copies that were previously upserted and that an ongoing search may still access.

We therefore first move to a slightly more low-level abstraction. To this end, we define the *upsert history* $H \subseteq \mathsf{KS} \times \mathbb{N}_\perp$ of a multicopy data structure as the set of all copies (k, t) that have been upserted thus far in the entire history of the computation. In particular, this means that any template algorithm will have to maintain the invariant $H \supseteq \bigcup_{n \in N} C_n$. We further define $\bar{H} := \lambda k.\ \max\{t \mid (k, t) \in H\}$. Since the data structure must always contain the latest copy of a key that was upserted, $\bar{H}$ coincides with the logical contents M.

Assume that, similar to $\overline{\mathsf{MCS}}(r, t, M)$, we are given a template-specific predicate $\mathsf{MCS}(r, t, H)$ that abstracts the state of a multicopy structure by its upsert history H and the current value t of the global clock. The desired atomic specification of `upsert` in terms of the new abstraction is simply:

$$\left\langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\right\rangle \texttt{ upsert r k } \left\langle\, \mathsf{MCS}(r, t+1, H \cup (k, t)) \,\right\rangle \tag{7.3}$$

To derive the atomic specification of `search`, first note that because H collects all copies of keys that have been upserted thus far, at the point when the returned timestamp t' of a search is determined, we must necessarily have $(k, t') \in H$ for the search to be correct. In addition, we need to capture that t' is either the logical timestamp of k at the start of the search or t' is a time point of an upsert for k that happened after the search started. To express this constraint, we assume an additional template-specific abstract predicate $\mathsf{mcs\_sr}(k, t_0)$. This predicate must hold at the invocation point of `search` and imply that t_0 is the logical timestamp of k at that point. The atomic specification of `search` that captures the search recency property is then as follows:

$$\begin{array}{l} \mathsf{mcs\_sr}(k, t_0) \mathrel{-\!\!*} \\ \quad \left\langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\right\rangle \texttt{ search r k } \left\langle\, t'.\ \mathsf{MCS}(r, t, H) * (k, t') \in H * t_0 \leq t' \,\right\rangle \end{array} \tag{7.4}$$

This new specification solves our problem of having to reason about non-fixed commit points when proving the atomic triple. That is, in the proof of `search`, we can now always commit the atomic triple at the point when the return value of the search is determined, i.e., when $(k, t') \in H$ is established. This point is independent of the concurrently executing upserts.

The following theorem relates the two types of specifications at the meta-level to obtain the desired overall correctness result. It is based on the observation that a concurrent execution of `upsert` and `search` operations that satisfy the atomic specifications (7.3) and (7.4) can be linearized by

letting the upserts in the equivalent sequential execution occur in the same order as their atomic commit points in the concurrent execution, and by letting each search $r\ k$ occur at the earliest time after the timestamp referenced by the returned copy t' of k. That is, if the timestamp t_0 associated with k at the time search begins is equal to t' then search occurs right after its invocation in the concurrent execution. Otherwise, if $t_0 < t'$, then it occurs right after t' was upserted.

Theorem 7.1 *Any multicopy structure whose operations satisfy the low-level atomic specifications is linearizable with respect to the high-level sequential specifications.*

Proof. Assume that we have a multicopy structure whose operations satisfy the atomic specifications (7.3) and (7.4). Assume further that the structure is initialized with root r satisfying abstract state $\mathsf{MCS}(r, 0, H_0)$ where $t_0 = 0$ and $H_0 = \{(k, \bot) \mid k \in KS\}$.

To relate the low-level atomic specifications with the high-level sequential specifications of the multicopy structure operations, we first define $\overline{\mathsf{MCS}}(r, t, M)$ in terms of $\mathsf{MCS}(r, t, H)$:

$$\overline{\mathsf{MCS}}(r, t, M) := \exists H.\, \mathsf{MCS}(r, t, H) * M = \bar{H}$$

Now consider a concurrent history of operations $\{op_i\ r\ k_i\}_{i\in[1,n]}$ for $n \geq 0$ with $op_i \in \{\mathtt{search}, \mathtt{upsert}\}$. Further for every $op_i = \mathtt{search}$, assume that $\mathsf{mcs\_sr}(k_i, t_{0,i})$ held at the point when $op_i\ r\ k_i$ was invoked and let t'_i be the return value of that search. Finally, assume that all operations are ordered by the time point when the unique commit point of their low-level atomic triples are reached (i.e., when the operation took effect). We show that this history can be linearized by appropriately reordering operations such that each operation satisfies the sequential specifications (7.1) and (7.2) after reordering.

First, it follows from the atomic specifications that, at the commit point of $op_i\ r\ k_i$, we must have $\mathsf{MCS}(r, t_i, H_i)$ where $t_i = t_{i-1}$ and $H_i = H_{i-1}$ if $op_i = \mathtt{search}$, respectively, $t_i = t_{i-1} + 1$ and $H_i = H_{i-1} \cup \{(t_{i-1}, k_i)\}$ if $op_i = \mathtt{upsert}$. Moreover, since only upserts modify the abstract state and the clock is incremented with every upsert, $op_i = \mathtt{upsert}$ implies $\max(H_{i-1})(k_i) < t_{i-1}$. In the following, we further let $M_i := \max(H_i)$ for all $i \in [1, n]$.

We first show that upserts do not need to be reordered relative to each other since they always satisfy their sequential specifications. That is, suppose $op_i = \mathtt{upsert}$ for some $i \in [1, n]$, then we

have:

$$\begin{aligned}
M_i &= \max(H_i) \\
&= \max(H_{i-1} \cup \{(k_i, t_{i-1})\}) \\
&= \lambda k.\, \max\{t \mid (k,t) \in H_{i-1} \cup \{(k_i, t_{i-1})\}\} \\
&= \lambda k.k = k_i \;?\; \max\{t \mid (k,t) \in H_{i-1} \cup \{(k_i, t_{i-1})\}\} : \\
&\qquad\qquad \max\{t \mid (k,t) \in H_{i-1} \cup \{(k_i, t_{i-1})\}\} \\
&= \lambda k.\, (k = k_i \;?\; t_{i-1} : \max\{t \mid (k,t) \in H_{i-1}\}) \qquad\qquad (\max(H_{i-1})(k_i) < t_{i-1}) \\
&= \lambda k.\, (k = k_i \;?\; t_{i-1} : \max(H_{i-1})(k)) \\
&= \max(H_{i-1})[k_i \rightarrowtail t_{i-1}] \\
&= M_{i-1}[k_i \rightarrowtail t_{i-1}]
\end{aligned}$$

Hence, the postcondition of (7.1) holds after op_i returns.

Now suppose $op_i = \texttt{search}$. First note that because searches do not modify the abstract state, they can be safely reordered without affecting the other operations. Next observe that, according to (7.4), we know that $t_{0,i} \leq t'_i$ and $t'_i \in H_i$. Hence, there must exist a unique op_j for some $j \in [1, i)$ that upserted k_i at time point t'_i. That is, $t_j = t'_i$, $k_j = k_i$ and therefore $H_{j+1} = H_j \cup \{(k_i, t'_i)\}$.

We distinguish two subcases. First, assume $t_j > t_{0,i}$, i.e., op_j took effect after $op_i\, r\, k_i$ was invoked. In this case, we can place op_i anywhere after op_j but before the next upsert operation in the sequence. We know that $\mathsf{MCS}(r, t_{j+1}, H_{j+1})$ holds before op_i is executed at its new place in the sequence. Moreover, since $\max(H_j)(k_i) < t_j$, we have $\max(H_{j+1})(k_i) = t_j = t'_i$ and hence $\mathsf{mcs\_sr}(k_i, t'_i)$ also holds. Now executing op_i sequentially yields, according to (7.4), some t' such that $(k_i, t') \in H_{j+1}$ and $t'_i \leq t'$. However, since (k_i, t'_i) is maximal in H_{j+1}, we must have $t'_i = t'$. Moreover, we have already established that $M_{j+1}(k_i) = \max(H_{j+1})(k_i) = t'_i$. Hence, the postcondition of (7.2) holds after op_i returns.

Now, assume $t_j = t_{0,i}$, i.e., op_j took effect before op_i was invoked. In this case, let $i' = \max\{j' \mid j' < i \wedge \max(H_{j'+1})(k_i) = t_{0,i}\}$. Note, that i' is well-defined since we have $j < i \wedge \max(H_{j+1})(k_i) = t_{0,i}$. We must necessarily have that $op_{i'}$ took effect either after op_i was invoked but before it returned, or it was the last operation that took effect before op_i was invoked. Either way, we can safely place op_i anywhere after $op_{i'}$ but before the next upsert in the sequence, respectively, before op_{i+1} if no upsert occurred between $op_{i'}$ and op_{i+1}. In all cases, $\mathsf{MCS}(r, t_{i'+1}, H_{i'+1})$ and $\mathsf{mcs\_sr}(k_i, t'_i)$ will hold before executing op_i, and we have $t'_i = M_{i'+1}(k_i)$ by construction. The proof that op_i must still return $t' = t'_i$ is analogous to the previous case. Hence, the postcondition of (7.2) holds again after op_i returns.

□

Maintenance operations. For a maintenance operation `m` (e.g. `flush` or `compact` for the LSM tree) the sequential and atomic specifications simply demand that the operation does not change the abstract state of the data structure:

$$\forall t\, M.\, \{\overline{\mathsf{MCS}}(r, t, M)\}\ \texttt{m r}\ \{\overline{\mathsf{MCS}}(r, t, M)\} \tag{7.5}$$

$$\langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\rangle\ \texttt{m r}\ \langle\, \mathsf{MCS}(r, t, H) \,\rangle \tag{7.6}$$

Since maintenance operations do not affect the abstract state of the data structure, Theorem 7.1 can be adapted to accommodate maintenance operations.

Summary. In order to prove the correctness of a given concurrent multicopy structure template, it thus suffices to show that each operation satisfies its corresponding atomic specification (7.3), (7.4), or (7.6)). In the following two chapters we discuss such proofs for two multicopy structure templates. The first template considers multicopy structures that consist of exactly two nodes and generalizes the differential file structures discussed in §7.2. It illustrates the basic RA constructions and invariants involved in such proofs without also suffering from the additional complexity of having to reason about an unbounded number of nodes. This template serves as a stepping stone to the second template, which considers the general case of an unbounded number of disk nodes and generalizes the LSM tree discussed in §7.3.

CHAPTER 8

Verifying the Two-Node Multicopy Template

In this chapter, we present a concurrent template for multicopy structures consisting of exactly one memory node and one disk node. The template, thus, generalizes from the differential file structures discussed in §7.2. We provide generic algorithms for the operations on such structures that abstract from the implementation-specific operations on the node level. We will then prove that these algorithms satisfy the atomic specifications of multicopy structures discussed in §7.4.

8.1 THE TWO-NODE MULTICOPY TEMPLATE

Figure 8.1 shows a potential abstract state of a two-node multicopy structure. The upsert operation is performed on the root node, while the search operation proceeds by first examining the root node for the operation key k, and moving on to the disk node if the root does not contain k. As there are only two nodes at all times and the disk node has unbounded size, maintenance is performed by a single operation that moves the data directly from the root node to the disk node. We next discuss these operations in more detail.

Multicopy operations. Figure 8.3 provides the algorithms for the operations on the two-node template. The operations are `search`, `upsert` and `reorganize`. These operations are defined in terms of implementation-specific helper functions whose formal specifications we will provide later. These are the functions `addContents`, `inContents` and `reorganizeContents`. The `upsert` additionally uses `readClock` and `incrementClock`. These auxiliary functions are *ghost code*, code added in order to facilitate the proof (these functions manipulate the ghost state that keeps track of the clock value) while having no effect on the program behavior. Technically, Iris does not require us to add such code in order to manipulate ghost state, but we use explicit ghost code here for the sake of clarity.

The `search` follows the general idea behind the data retrieval operations on differential file structures. The `search r d k` operation first locks the root node r and checks its contents. If a copy of key k is found, then it is returned after unlocking r. Otherwise, the `search` moves onto the disk node d and repeats the procedure. The `search` uses the helper function `inContents n k` to check if any copy of k is contained in node n. Note that `search`, like the link template from Chapter 6, does not hold locks when moving from one node to the next.

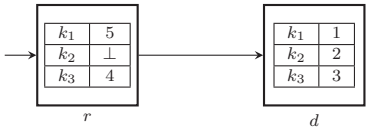

Figure 8.1: Abstract state of a two-node multicopy structure representing the differential file structure in Fig 7.1.

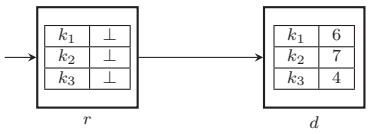

Figure 8.2: A reorganization of data on the two-node multicopy structure shown in Fig 8.1

The `upsert r k` operation locks the root node and adds a new version of the key k to the contents of the root node using `addContents`. In order to add a new copy of k, it must know the current clock value. This is accomplished by using the `read_clock ()` function. `addContents r k t` adds the pair (k, t) to the root node when it succeeds. `upsert` terminates by incrementing the clock value and unlocking the root node. The `addContents` function may however fail if the root node is full. In this case `upsert` calls itself recursively [1].

The `reorganize` operation is again inspired by the maintenance operation undertaken by differential file structures. The data structure simply locks both nodes and then merges the data from the root node into the disk node using the implementation-specific helper function `reorganizeContents`. If a copy of any key k is present in both the root node and the disk node, then the copy in the root node is preserved while the copy in the disk node is deleted. Figure 8.2 illustrates the result of this operation on the two-node multicopy structure shown in Fig. 8.1.

8.2 CORRECTNESS PROOF FOR THE TWO-NODE TEMPLATE

We now focus on proving the correctness of the two-node template algorithms. Recall Theorem 7.1, which states that a multicopy structure is linearizable if its operations satisfies the atomic specifications (7.3), (7.4), and (7.6). We adapt these specification to the two-node template:

$$\langle\, t\, H.\ \mathsf{MCS}(r,t,H)\,\rangle\ \texttt{search r d k}\ \langle\, t'.\ \mathsf{MCS}(r,t,H) * (k,t') \in H * t_0 \le t'\,\rangle$$
$$\langle\, t\, H.\ \mathsf{MCS}(r,t,H)\,\rangle\ \texttt{upsert r k}\ \langle\, \mathsf{MCS}(r,t+1,H \cup (k,t))\,\rangle$$
$$\langle\, t\, H.\ \mathsf{MCS}(r,t,H)\,\rangle\ \texttt{reorganize r d}\ \langle\, \mathsf{MCS}(r,t,H)\,\rangle$$

[1] For simplicity of presentation, we assume that a separate maintenance thread invokes `reorganize` to ensure that upserts eventually make progress. Also, note that we use a tail-recursive call, which ensures that the code will run in constant stack space.

```
1  let search r d k =
2    lockNode r;
3    let t' = inContents r k in
4    if t' != ⊥ then begin
5      unlockNode r; t'
6    end
7    else begin
8      unlockNode r;
9      lockNode d;
10     let t' = inContents d k in
11     unlockNode d; t'
12   end
13
14 let rec upsert r k =
15   lockNode r;
16   let t = readRlock() in
17   let res = addContents r k t in
18   if res then begin
19     incrementClock();
20     unlockNode r
21   end
22   else begin
23     unlockNode r;
24     upsert r k
25   end
26 let reorganize r d =
27   lockNode r;
28   lockNode d;
29   reorganizeContents r d;
30   unlockNode r;
31   unlockNode d
```

Figure 8.3: The two-node multicopy template algorithms. The template can be instantiated by providing implementations of helper functions `inContents`, `addContents` and `reorganizeContents`. `inContents n k` returns $C_n(k)$. `addContents r k t` adds the key k with the given clock value t to the contents of `root`. The return value of `addContents` is a Boolean which indicates whether the insertion was successful (e.g. if root node is full, insertion may fail leaving the contents unchanged).

Proving the atomic triple for `upsert` presents a minor technical challenge. The issue arises because we cannot access the atomic precondition once the operation has been linearized. [2] The linearization point of `upsert` is at line 22 in Fig. 9.1. However, after this point in the proof, we still need to access to resources associated with the data structure in order establish the precondition for unlocking the root node. If the atomic preconditions governs all the relevant resources, then we can no longer return the resources protected by the lock back to the atomic precondition at this point.

The solution here is to use invariants. An invariant in Iris is a formula of the form $\boxed{P}^{\mathcal{N}}$, where P is an arbitrary Iris formula. Invariants provide an orthogonal mechanism to atomic triples in order

[2] Recall from the discussion in §3.3.3 that the proof of an atomic triple proceeds by obtaining an atomic update token $\mathsf{AU}_{P,Q}$ (using rule LOGATOM-INTRO). Here P is the precondition of the atomic triple while Q is the postcondition. A thread possessing $\mathsf{AU}_{P,Q}$ can access resources in P before each atomic step. At the end of the atomic step, the thread must either establish P again or use P to generate Q (i.e. when the operation is linearized). In the latter case, the thread loses the ownership of $\mathsf{AU}_{P,Q}$. After this point, it can no longer access resources in P.

to reason about ownership of resources describing shared state that can be concurrently accessed by many threads. Intuitively, an invariant is a property that, once established, will remain true forever. It is therefore a duplicable resource and can be freely shared with any thread.

However, in order to ensure that the invariant indeed remains valid once it has been established, Iris' proof rules for invariants impose restrictions on how the resources contained in an invariant can be accessed and manipulated. At any point in time, a thread can *open* an invariant $\boxed{P}^{\mathcal{N}}$ and gain ownership of the contained resources P. These resources can then be used in the proof of a single atomic step of the thread's execution. After the thread has performed an atomic step with an open invariant, the invariant must be *closed*, which amounts to proving that P has been reestablished. Otherwise, the proof cannot succeed. In this sense, invariants behave much like the preconditions of atomic triples before the atomic triple has been committed. However, unlike atomic preconditions, an invariant is always accessible as long as it is reestablished after each atomic step.

The $\mathcal{N}$ in $\boxed{P}^{\mathcal{N}}$ refers to the *namespace* of the invariant. Namespaces are part of the mechanism used in Iris to keep track of invariants that are currently open and need to be closed before the next atomic step. This is necessary to avoid issues of re-entrancy in case of nested invariants, which would lead to logical inconsistencies. In this book, we do not use more than one invariant at any point in time. Hence, we omit the namespace annotations in the following. For a more in-depth discussion of Iris' invariant mechanism and the relevant proof rules, we refer the interested reader to [Jung et al., 2018].

With the invariant $\mathsf{mcs\_inv}(r, d)$ describing the shared state of the two-node multicopy structure, we amend the specifications of `search`, `upsert` and `reorganize` as follows:

$$\boxed{\mathsf{mcs\_inv}(r,d)} \mathrel{-\!\!*} \mathsf{mcs\_sr}(k,t_0) \mathrel{-\!\!*} \left\langle t\,H.\ \mathsf{MCS}(r,t,H) \right\rangle \texttt{search r d k} \left\langle t'.\ \mathsf{MCS}(r,t,H) * (k,t') \in H * t_0 \le t' \right\rangle$$

$$\boxed{\mathsf{mcs\_inv}(r,d)} \mathrel{-\!\!*} \left\langle t\,H.\ \mathsf{MCS}(r,t,H) \right\rangle \texttt{upsert r k} \left\langle \mathsf{MCS}(r,t+1,H \cup (k,t)) \right\rangle$$

$$\boxed{\mathsf{mcs\_inv}(r,d)} \mathrel{-\!\!*} \left\langle t\,H.\ \mathsf{MCS}(r,t,H) \right\rangle \texttt{reorganize r d} \left\langle \mathsf{MCS}(r,t,H) \right\rangle$$

An atomic triple guarded by an invariant can be interpreted as satisfying the atomic triple under the assumption that the shared state satisfies the invariant. In particular, the invariant $\mathsf{mcs\_inv}(r, d)$ relates the upsert history H used in the specifications to the contents of the two nodes of the data structure. We provide its definition as well as the definition of the predicate $\mathsf{mcs\_sr}(k, t_0)$ used to snapshot the current logical timestamp t_0 of key k in the next section.

8.2.1 PROVING SEARCH RECENCY

We start with the proof of `search`.

High-level proof idea. Recall that search recency says that if t_0 is the logical timestamp of k at the point when `search r d k` is invoked, then the operation returns $(k, t) \in H$ such that $t \geqslant t_0$. Since the timestamp t of the copy (and, in the full implementation, the value) of k retrieved by `search`

either comes from the root or disk node, we must examine the relationship between the history-dependent contents H of the data structure and the physical contents C_n of the nodes n visited as the search progresses. We do this by identifying the key invariants needed for proving search recency in general for multicopy structures and then proceed to show that these are satisfied by the two-node multicopy template.

We refer to the *spatial ordering* of the copies (k, t) stored in a multicopy structure as the ordering in which these copies are reached when traversing the data structure graph starting from the root node. Our first observation is that the spatial ordering is consistent with the temporal ordering in which the copies have been upserted. In other words, the farther away a search gets from the root node, the older the copies are that it will find. Therefore, when a search traverses the data structure without being interfered with by other threads and it returns the first copy of its query key k that it finds, it is guaranteed to return the current logical timestamp of k.

We formalize this observation in terms of the *contents-in-reach* of a node. The contents-in-reach of a node n is the function $C_{ir}(n) : \mathsf{KS} \to \mathbb{N}_\perp$ defined recursively over the graph of the multicopy structure as follows:

$$C_{ir}(n)(k) \coloneqq \begin{cases} C_n(k) & \text{if } C_n(k) \neq \perp \\ C_{ir}(n')(k) & \text{if } k \in \mathsf{es}(n, n') \text{ for some } n' \in N \\ \perp & \text{otherwise} \end{cases} \tag{8.1}$$

Note that $C_{ir}(n)$ is well-defined because the graph is acyclic and the edgesets labeling the outgoing edges of every node n are disjoint.

For the simple case of a two-node multicopy structure with root node r and disc node d, the above definition simplifies to:

$$C_{ir}(r) = \lambda k.\ (C_r(k) \neq \perp\ ?\ C_r(k) : C_d(k))$$
$$C_{ir}(d) = C_d(k)$$

That is, k is in the contents-in-reach of the root if it is in the root; otherwise, k is in contents-in-reach of the root if it is in the contents of the disk node.

The observation that interference-free searches will find the current logical timestamp of their query key is then captured by the following invariant:

Invariant 1 At every atomic step, the logical contents of the multicopy structure is the contents-in-reach of its root node: $\bar{H} = C_{ir}(r)$.

In order to account for concurrent threads interfering with the search, we prove the condition $t_0 \leq t'$ for the timestamp t' returned by the search. Intuitively, this is true because the contents-in-reach of a node n can be affected only by upserts or maintenance operations, both of which only increase the timestamps associated with keys: upserts insert new copies into the root node and maintenance operations move recent copies down in the structure, possibly replacing older copies. This observation is formally captured by the following invariant:

Invariant 2 From one atomic step to the next, the (timestamp) contents-in-reach of every node can only increase. That is, for every node n and key k, if $C_{ir}(n)(k) = t$ at some point in time and $C_{ir}(n)(k) = t'$ at any later point in time, then $t \leq t'$.

Finally, in order to prove the condition $(k, t') \in H$ of search recency, we need one additional property:

Invariant 3 At every atomic step, all copies present in the multicopy structure have been upserted at some point in the past. That is, for all nodes n, $C_n \subseteq H$.

Now let us consider an execution of `search` on k. If `search` finds a copy $t' = C_r(k)$ with $t' \neq \bot$ in the root node, then by Invariant 1 and the definition of contents-in-reach, we can immediately conclude $t' = C_{ir}(r)(k) = \bar{H}(k)$. Moreover, Invariant 3 implies $(k, t') \in H$. Finally, we know that at the time of invocation of `search` we had $\bar{H}(k) = t_0$. Again, we know from Invariant 1 that we also had $\bar{H}(k) = C_{ir}(r)(k)$ at that point. Hence, Invariant 2 implies $t_0 \leq t'$.

On the other hand, if $C_r(k) = \bot$, then it follows from Invariant 1 that we must have had $\bar{H}(k) = C_{ir}(d)(k) = t_1$ for some value t_1 at the point when r was accessed. Moreover, Invariant 2 implies again $t_0 \leq t_1$. The search now proceeds to the disk node and will return t' such that $t' = C_d(k) = C_{ir}(d)(k)$ holds at this point. While t' may differ from t_1, Invariant 2 guarantees again that $t_1 \leq t'$ and hence $t_0 \leq t'$. Moreover, Invariant 3 ensures $(k, t') \in H$. Hence, search recency holds in this case as well.

In the remainder of this subsection we explain how to formalize this high-level proof in Iris.

Invariant for the two-node multicopy structure. We start with the Iris invariant $\mathsf{mcs\_inv}(r, d)$ that we will be using in this proof. Figure 8.4 shows the parts of the invariant needed to formally capture the high-level invariants 1 to 3 for proving search recency. We ignore some parts that are irrelevant for now such as the ghost resources needed to keep track of the global clock. We will present the full invariant when we discuss the proof of `upsert`.

We explain each part of the invariant in detail:

- The existentially quantified variables t, H, C_r and C_d are to be interpreted as the current clock value, the current upsert history, the current contents of the root node and the current contents of the disk node, respectively.
- As is standard when using logically atomic triples in combination with invariants in Iris, we introduce two predicates capturing the state of the data structure, one owned by the invariant ($\mathsf{MCS}^\bullet(r, t, H)$) and one used in the atomic triples ($\mathsf{MCS}(r, t, H)$). We can think of these predicates as providing two different views at the logical state of the data structure, one from the perspective of the data structure's implementation, and one from the perspective of the client. Together, they provide the following important properties:

VIEW-UPD
$$\frac{\mathsf{MCS}^\bullet(r, t, H) * \mathsf{MCS}(r, t, H)}{\mathsf{MCS}^\bullet(r, t', H') * \mathsf{MCS}(r, t', H')}$$

VIEW-SYNC
$$\mathsf{MCS}^\bullet(r, t, H) * \mathsf{MCS}(r, t', H') \vdash t = t' \land H = H'$$

$$
\begin{aligned}
\mathsf{mcs\_inv}(r,d) \coloneqq \exists\, t, H, C_r, C_d.\ & \\
& \mathsf{MCS}^\bullet(r,t,H) \\
& * \boxed{\bullet\, H}^{\gamma_s} \\
& * (\exists b_r.\ \mathsf{lk}(r) \mapsto b_r * (b_r\ ?\ \mathsf{True} : \mathsf{N}(r,r,C_r))) * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} \\
& * (\exists b_d.\ \mathsf{lk}(d) \mapsto b_d * (b_d\ ?\ \mathsf{True} : \mathsf{N}(r,d,C_d))) * \boxed{\tfrac{1}{2} C_d}^{\gamma_{c(d)}} \\
& * \mathsf{cir}(H, C_r, C_d) \\
& * \mathop{\Large *}_{k \in \mathsf{KS}} \boxed{\bullet\, C_d(k)}^{\gamma_{d(k)}}
\end{aligned}
$$

where $\mathsf{N}(r,n,C_n) \coloneqq \mathsf{node}(r,n,C_n) * \boxed{\circ\, C_n}^{\gamma_s} * \boxed{\tfrac{1}{2} C_n}^{\gamma_{c(d)}}$

$$\mathsf{cir}(H, C_r, C_d) \coloneqq \bar{H} = \lambda k.\ (C_r(k) \neq \bot\ ?\ C_r(k) : C_d(k))$$

Figure 8.4: The invariant for the two-node multicopy template.

The rule VIEW-UPD says that both predicates are required in order to update the views of the data structure. This occurs when both the invariant as well as the precondition of an atomic triple are accessed at an atomic step. The rule VIEW-SYNC ensures that both copies are always in sync. The predicates $\mathsf{MCS}^\bullet(r,t,H)$ and $\mathsf{MCS}(r,t,H)$ are defined using a combination of authoritative, option and exclusive RAs.

- We use the authoritative RA of sets of key/timestamp pairs, $\textsc{Auth}(\mathsf{KS} \times \mathbb{N}_\bot)$, at ghost location γ_s to keep track of the upsert history. That is, $\boxed{\bullet\, H}^{\gamma_s}$ expresses the fact that the current upsert history is H. Recall that the only frame-preserving update admitted by authoritative sets is to add more elements to the set, which is consistent with the fact that H can only grow.

- The predicates $\mathsf{N}(r,n,C_n)$ contain the resources guarded by the lock bit $\mathsf{lk}(n)$ of node n. In particular, this includes the implementation-specific predicate $\mathsf{node}(r,n,C_n)$, which captures ownership of the resources needed to implement the node n. The predicate $\mathsf{node}(r,n,C_n)$ must also tie C_n to the implementation-specific physical representation of the node's contents. The third and forth conjuncts of the invariant thus capture the fact that a thread cannot modify the contents of r or d without locking them first. We parameterize node by the root node r in order to allow the implementation to choose different representations for memory and disk nodes and optimize the node-level operations for the underlying storage medium (this can be done by distinguishing between $r = n$ and $r \neq n$). Further, note that the assertion $\boxed{\circ\, C_n}^{\gamma_s}$ in $\mathsf{N}(r,n,C_n)$ together with $\boxed{\bullet\, H}^{\gamma_s}$ imply that $C_n \subseteq H$. This gives us Invariant 3.

- The fractional resources stored at ghost locations $\gamma_{c(r)}$ and $\gamma_{c(d)}$ ensure that when $n \in \{r, d\}$ is locked and ownership of $\mathsf{N}(r, n, C_n)$ is transferred from the invariant to the thread locking n, then the C_n that is quantified in the invariant remains tied to the actual physical contents of the node. This is achieved by splitting ownership of C_n in $\gamma_{c(n)}$ between $\mathsf{N}(r, n, C_n)$ and the invariant.

- The predicate $\mathsf{cir}(H, C_r, C_d)$ directly captures Invariant 1, relating the logical contents with the contents-in-reach of the root node.

To see that Invariant 2 is also captured, first note that the authoritative set RA used for the ghost location γ_s storing H enforces that H can only grow over time. It follows by definition of $\bar{H}$ that $\bar{H}$ can only increase. Thus Invariant 1 implies that Invariant 2 holds for r. We use the final conjunct of the invariant to capture that Invariant 2 also holds for d. For this purpose, we use the authoritative RA of natural numbers with maximum as the underlying monoid operation (referred to from now on as the authoritative mnat RA). The authoritative maxnat RA guarantees the following properties:

$$\textsc{auth-maxnat-valid}\ \frac{\mathcal{V}(\bullet m \cdot \circ n)}{m \geqslant n} \qquad \textsc{auth-maxnat-upd}\ \frac{m \leqslant n}{\bullet m \rightsquigarrow \bullet n} \qquad \textsc{auth-maxnat-snap}\ \ \bullet m \rightsquigarrow \bullet m \cdot \circ m$$

The ghost resource $\bullet\, m$ signifies that the current value is m and the ghost resource $\circ\, n$ expresses that n is a lower bound on the current value. That is, $\bullet m \cdot \circ n$ can be valid only if $m \geqslant n$, as captured by rule AUTH-MAXNAT-VALID. Consequently, the only frame-preserving update permitted by this RA is to replace the current value m by any larger value (rule AUTH-MAXNAT-UPD). Finally, rule AUTH-MAXNAT-SNAP allows us to take a snapshot of the current value m and remember it as a lower bound for a value of the resource that is observed at a later point in the proof.

In order to express Invariant 2 for d, the last conjunct in $\mathsf{mcs\_inv}(r, d)$ uses for every key k a ghost location $\gamma_a(k)$ that stores the current value $C_d(k)$ as an authoritative maxnat. The RA will then enforce that $C_d(k)$ can only increase over time.

Capturing the search recency property. Finally, we need to define the predicate $\mathsf{mcs\_sr}(k, t_0)$ which we use in the specification of search recency, the key property allowing us to reorder searches in the meta-level linearizability proof (Theorem 7.1). Ideally, we would like this predicate to express the fact that t_0 was the logical timestamp of k at the precise point when `search` was invoked. Unfortunately, there is no easy way to express this fact when reasoning modularly about the implementation of `search` without constraining the client that calls `search`. So instead we proceed as follows. We define $\mathsf{mcs\_sr}(k, t_0)$ to capture the fact that t_0 was the logical timestamp of k at some point in the past (relative to the time point when the predicate holds). By requiring $\mathsf{mcs\_sr}(k, t_0)$ in the precondition of `search`, we thus state that t_0 was the logical timestamp of k at some point prior to the invocation of `search`. Since the specification universally quantifies over t_0, we will prove that search recency holds for all such values t_0, including the logical timestamp of k at the actual point

of invocation. Note that the resulting specification is in fact equivalent to the desired one because the logical timestamp of a key only increases over time and the postcondition of `search` requires $t_0 \leq t'$ for the returned time stamp t'. That is, if $t_0 \leq t'$ holds for the logical timestamp t_0 at the point of invocation of `search`, then for all prior logical timestamps t_0' of k before k was upserted to t_0, we must also have $t_0' \leq t'$.

In order to define the predicate $\mathsf{mcs\_sr}(k, t_0)$, we reuse the ghost location γ_s, which stores the current history-dependent contents H in an authoritative set RA. Recall from the discussion of the authoritative set RA that the assertion $\boxed{\bullet H}^{\gamma_s} * \boxed{\circ H'}^{\gamma_s}$ expresses the fact that H is the current authoritative version of the set at ghost location γ_s and $H' \subseteq H$. Since $\boxed{\bullet H}^{\gamma_s}$ is part of the invariant $\mathsf{mcs\_inv}(r)$, the assertion $\mathsf{mcs\_inv}(r) * \boxed{\circ \{(k, t_0)\}}^{\gamma_s}$ implies $(k, t_0) \in H$. In other words, if $\mathsf{mcs\_inv}(r)$ and $\boxed{\circ \{(k, t_0)\}}^{\gamma_s}$ hold simultaneously, then (k, t_0) must have been upserted at some point in the past and t_0 was the logical timestamp of k from that point onward until the next upsert of k. We therefore define:

$$\mathsf{mcs\_sr}(k, t_0) :\Leftrightarrow \boxed{\circ \{(k, t_0)\}}^{\gamma_s}$$

Proof of `search`. The outline of the formal correctness proof for `search` is shown in Fig. 8.5. The proof assumes that the implementation-specific helper function `inContents` satisfies the specification given on line 1. We also provide logically atomic triples that specify the assumed behavior of the functions for locking and unlocking the nodes (at lines 2 and 3). The function `lockNode` atomically produces $\mathsf{N}(r, n, C_n)$ using the invariant, while the function `unlockNode` consumes the $\mathsf{N}(r, n, C_n)$ (by assimilating it back into the invariant).

Let us now turn to the outline of the proof. The intermediate assertions shown in Fig 8.5 represent the relevant information from the proof context at the corresponding point. By convention, all the newly introduced variables are existentially quantified. Note that the condition $\mathsf{mcs\_sr}(k, t_0)$ is persistent and, hence, hold throughout the proof. Moreover, the invariant $\boxed{\mathsf{mcs\_inv}(r, d)}$ is maintained throughout the proof since `search` does not modify any shared resources. We therefore do not include these resources explicitly in the intermediate assertions.

The search operation starts by locking the root node r. This gives us access to the predicate $\mathsf{N}(r, r, C_r)$ at line 8. Next, we apply the specification of `inContents` to the root node r. This gives rise to two cases: (i) The call returns $C_r(k) = t' \neq \bot$ or (ii) the call returns $C_r(k) = t' = \bot$.

Let us consider the first case (line 12). At this point we have found the logical timestamp of k and we can commit the atomic triple when the thread unlocks r in line 16. To prove the postcondition of the atomic triple, we first obtain its precondition $\mathsf{MCS}(r, t, H)$. Opening the invariant and using rule VIEW-SYNC we can conclude that H is the current authoritative upsert history. Hence, using $\mathsf{cir}(H, C_r, C_d)$ from the invariant and $C_r(k) \neq \bot$ we can now infer $\bar{H}(k) = C_r(k)$. Moreover, using $\mathsf{mcs\_sr}(k, t_0)$ we can conclude $t_0 \leq \bar{H}(k)$ (line 14). Thus, we must have $\bar{H}(k) = t'$, which

gives us $(k, t') \in H$ and $t_0 \leq t'$ (line 15). Unlocking r transfers $\mathsf{N}(r, r, C_r)$ back to the invariant, which leaves us with the postcondition of the atomic triple. This completes case (i).

Now we consider the second case at line 20, where $C_r(k) = \bot$. In this case, we unlock the root node and move on to the disk node. But before we unlock, we can infer useful information from the fact that $C_r(k) = \bot$. First, using $\mathsf{cir}(H, C_r, C_d)$ from the invariant, we know that $\bar{H}(k) = C_d(k)$. Moreover, using $\mathsf{mcs\_sr}(k, t_0)$ we conclude $t_0 \leq \bar{H}(k)$, and thus $t_0 \leq C_d(k)$ (line 21). Next, we use rule AUTH-MAXNAT-SNAP to take a snapshot of $\boxed{\bullet\ C_d(k)}^{\gamma_{d(k)}}$ in the invariant to obtain $\boxed{\circ\ C_d(k)}^{\gamma_{d(k)}}$. As we won't be able to maintain $C_d(k)$ once r has been unlocked, we replace it by a fresh variable t_1 (line 22). Now the thread unlocks r, returning ownership of $\mathsf{N}(r, r, C_r)$ back to the invariant and immediately locks d, which transfers ownership of $\mathsf{N}(r, d, C_d)$ from the invariant to the thread (line 26). At this point, we open the invariant and use the rule AUTH-MAXNAT-VALID to conclude that $t_1 \leq C_d(k)$. Hence, we also have $t_0 \leq C_d(k)$ (line 27).

The thread now uses `inContents` to retrieve the current value t' of k in d (i.e. $C_d(k)$), which it will return once it has unlocked d on line 33. This is going to be the commit point for the atomic triple in case (ii). Hence, we access the precondition $\mathsf{MCS}(r, t, H)$ of the atomic triple before the call to `unlockNode` (line 31). To obtain the postcondition of the atomic triple, we open the invariant and use rule VIEW-SYNC to conclude that H is the current authoritative upsert history. Moreover, from $\boxed{\circ\ C_d}^{\gamma_s}$ in $\mathsf{N}(r, d, C_d)$ we infer $(k, C_d(k)) \in H$. Replacing $C_d(k)$ with t' we arrive at line 32. Finally, unlocking d transfers $\mathsf{N}(r, d, C_d)$ back to the invariant, which leaves us with the postcondition of the atomic triple. This completes case (ii).

8.2.2 PROVING THE CORRECTNESS OF UPSERT

We next verify the algorithm of `upsert` provided in Fig. 8.3 against its atomic specification:

$$\mathsf{mcs\_inv}(r, d) \twoheadrightarrow \left\langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\right\rangle\ \texttt{upsert r k}\ \left\langle\, \mathsf{MCS}(r, t+1, H \cup (k, t)) \,\right\rangle$$

Unlike the search operation, the upsert operation exhibits a fixed linearization point. The linearization point is when the `addContents` operation succeeds. The proof is nevertheless interesting as we need to show that `upsert` maintains the invariants used for the proof of `search` in the previous section.

High-level proof idea. In order to prove the logically atomic specification for `upsert` described previously, we must identify an atomic step where the clock value is incremented and the upsert history is updated. Intuitively, this atomic step is when the global clock is incremented (line 20 in Fig 8.3) after `addContents` succeeds. Note that in this case `addContents` changes the contents of the root node from C_r to $C_r' = C_r[k \rightarrowtail t]$. Hence, in the proof we need to update the upsert history from H to $H' = H \cup \{(k, t)\}$, reflecting that the new copy has been upserted. It then remains to show that the three key high-level invariants of multicopy structures are preserved by these updates. We argue more generally, that these invariants are maintained by an upsert on the root node of any multicopy structure.

First, observe that Invariant 3, which states $C_n \subseteq H$, is trivially maintained: only C_r is affected by the upsert and the new copy (k, t) is included in H'. Similarly, we can easily show that Invariant 2 is maintained: $C_{ir}(n)$ remains the same for all nodes $n \neq r$ and for the root node it increases, provided Invariant 1 is also maintained.

Thus, the interesting case is Invariant 1. Proving that this invariant is maintained amounts to showing that $\max[H'](k) = t$. This step critically relies on the following additional observation:

Invariant 4 At every atomic step, all timestamps in H are smaller than the current time of the global clock t.

This invariant implies that $\bar{H}'(k) = \max(\bar{H}(k), t) = t$, which proves the desired property. We note that Invariant 4 is maintained because the global clock is incremented in the same step when H is updated to H'.

In the proof of Invariant 1 we have silently assumed that the timestamp t, which was obtained by reading the global clock at 16, is still the value of the global clock at the linearization point when the clock is incremented at 19. This step in the proof relies on the observation that only `upsert` changes the global clock and it does so only while the clock is protected by the root node's lock. Hence, for lock-based implementations of multicopy structures, we additionally require the following invariant:

Invariant 5 From one atomic step to the next, if a thread holds the lock on the root node, no other thread will change the value of the global clock.

In the remainder of this section, we show how to formalize this high-level proof in Iris.

Updated invariant for the two-node multicopy structure. We first update $\mathsf{mcs\_inv}(r, t, H)$ to capture the additional high-level invariants required by the proof of `upsert`. The new Iris invariant is shown in Fig 8.6. The changes are as follows:

- We track the current value t of the global clock in a fractional authoritative maxnat RA at ghost location γ_t. The RA ensures that the clock value can only increase. To capture Invariant 5, we split this resource half-way between the invariant and the predicate $\mathsf{N}(r, r, C_r)$, which is protected by the lock of the root node.
- We add the constraint $\mathsf{maxTS}(t, H)$, which expresses Invariant 4.

Note that the new invariant implies the old invariant in Fig. 8.4 that we used for the proof of `search`. Since `search` does not modify any of the ghost resources, the proof also works with the new invariant.

Proof of `upsert`. Figure 8.7 shows the outline of the proof of `upsert`. The first line provides the specification of the implementation-specific helper function `addContents` that we assume in the proof. The specification simply says that when the function succeeds, the copy of k is updated to t in C_r. In case it fails, then no changes are made. We also provide specifications of the ghost code

functions `readClock` and `incrementClock` for manipulating the ghost resource for the global clock at ghost location γ_t. The function `readClock` requires fractional ownership of the clock resource for some non-zero fraction q to read the current clock value (line 2). On the other hand, `incrementClock` needs full ownership of the resource to increment the current clock value (line 3). Note that the functions `lockNode` and `unlockNode` follow the same specification as in the proof of `upsert` (see Fig. 8.5).

With everything needed for the proof of `upsert` in place, let us now walk through the proof outline shown in Fig. 8.7. We start with the invariant $\boxed{\mathsf{mcs\_inv}(r, d)}$ and atomic precondition $\langle\, t\, H.\ \mathsf{MCS}(r, t, H)\,\rangle$. The invariant can be accessed at each atomic step, but must also be reestablished after each step. Similarly, the atomic precondition is accessible at each atomic step, and must either be used to generate the postcondition of the atomic triple or the precondition must be reestablished.

The thread first locks the root node, which transfers ownership of $\mathsf{N}(r, r, C_r)$ from the invariant to the thread (line 8). At this point, we unfold the definition of $\mathsf{N}(r, r, C_r)$ (line 9) as we will need the contained resources later in the proof. We here use the variable t_1 to refer to the current value of the global clock. Next, the thread uses `readClock` to read the current clock value into the local variable t. We can use the fractional permission $\boxed{\tfrac{1}{2} \bullet t_1}^{\gamma_t}$ and the specification of `readClock` to conclude $t = t_1$ (line 11). The thread now calls `addContents` to update r with the new copy of k for timestamp t. This leaves us with two possible scenarios depending on whether the return value `res` is `True` (line 14) or `False` (line 25).

In the case where `addContents` fails, no changes have been performed. Here, we simply fold the definition of $\mathsf{N}(r, r, C_r)$ again, unlock r to transfer ownership of the node's resources back to the invariant and simply commit the atomic triple on the recursive call to `upsert`.

In the case where `addContents` succeeds, we obtained $C'_r = C_r[k \rightarrowtail t]$ from its postcondition, where C'_r is the new contents of the root node. The thread will next call `incrementClock` to increase the global clock value. This will be the linearization point of this branch of the conditional expression. Hence, we will also have to update all remaining ghost resources to their new values at this point. To prepare committing the atomic triple, we first access its precondition to obtain $\mathsf{MCS}(r, t', H')$ for some fresh variables t' and H' (line 16). We then open the invariant to access $\mathsf{MCS}^\bullet(r, t, H)$ and use rule VIEW-SYNC to conclude $H' = H$ and $t' = t$ (line 17).

The actual commit of the atomic triple involves several steps. First, the thread calls `incrementClock`. To satisfy the precondition of `incrementClock`, we open the invariant to retrieve the second half of the fractional clock resource at ghost location γ_t, and combine it with the half in the thread's local state to obtain the full resource $\boxed{\bullet t}^{\gamma_t}$. The postcondition of `incrementClock` then gives us back $\boxed{\bullet t + 1}^{\gamma_t}$ which we split again into two halves, returning one half to the invariant and keeping the other in the local proof context. In addition, we update all remaining ghost resources:

- We update the authoritative version of the history dependent state at ghost location γ_s in the invariant from H to $H' = H \cup \{(k, t)\}$, using rule AUTH-SET-UPD.

- We use rule FRAC-UPD to update the resource holding the root's contents at location $\gamma_{c(r)}$ from C_r to C'_r by reassembling the full resource from the half owned by the invariant, respectively, the half owned by the local proof context. After the update, the resource is split again into two halves, with one half returned to the invariant.
- We use rule VIEW-UPD to update the client's and invariant's views of the data structures state to $\mathsf{MCS}(r, t+1, H')$ and $\mathsf{MCS}^\bullet(r, t+1, H')$, respectively.

This leaves us with the new proof context shown on line 19. It remains to show that the updates of the ghost resources preserve the invariant. That is, we need to prove that $\mathsf{maxTS}(t+1, H')$ and $\mathsf{cir}(H', C'_r, C_d)$ hold. First note that $\mathsf{maxTS}(t+1, H')$ follows directly from the definition of H' and $\mathsf{maxTS}(t, H)$. We obtain the later from the invariant prior to the call to `incrementClock`. To show $\mathsf{cir}(H', C'_r, C_d)$, we need to prove that for all keys k'

$$\bar{H}'(k') = (C'_r(k') \neq \bot \ ?\ C'_r(k') : C_d(k'))$$

If $k' \neq k$, the equality follows directly from $\mathsf{cir}(H, C_r, C_d)$ and the definitions of H' and C'_r. For the case where $k' = k$ observe that $\mathsf{maxTS}(t, H)$ implies $\bar{H}'(k) = t$. Moreover, we have $C'_r(k) = t$ by definition of C'_r and we also know $t \neq \bot$ because the clock resource can only hold natural numbers.

Finally, in order to prove that we can safely unlock r, we have to reassemble $\mathsf{N}(r, r, C'_r)$ from the proof context at line 19. We have all the relevant pieces available, except for $\boxed{\circ\, C'_r}^{\gamma_s}$. We obtain this remaining piece by observing that $\boxed{\circ\, C_r}^{\gamma_s}$ implies $C_R \subseteq H'$ which in turn implies $C'_r \subseteq H'$ by definition of C'_r and H'. Using rule AUTH-SET-SNAP we obtain $\boxed{\circ\, H'}^{\gamma_s}$, which we can rewrite to $\boxed{\circ\, (H' \cup C'_r)}^{\gamma_s}$ using the previously derived equality $H' = H' \cup C'_r$. Applying rule AUTH-FRAG-OP, we can then infer $\boxed{\circ\, H'}^{\gamma_s} * \boxed{\circ\, Cr'}^{\gamma_s}$ and after throwing away the first conjunct, we are left with the desired missing piece.

After reassembling $\mathsf{N}(r, r, C'_r)$ we arrive at line 20 at which point the thread unlocks r, transferring $\mathsf{N}(r, r, C'_r)$ back to the invariant and this concludes the proof of `upsert`.

8.2.3 PROVING THE CORRECTNESS OF MAINTENANCE

Finally, we show that `reorganize` satisfies the following atomic specification, as required by Theorem 7.1:

$$\boxed{\mathsf{mcs\_inv}(r, d)} \twoheadrightarrow \left\langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\right\rangle \texttt{ reorganize r d } \left\langle\, \mathsf{MCS}(r, t, H) \,\right\rangle$$

Proving this atomic specification is simpler than proving the corresponding specifications for search and upsert, because `reorganize` (exclusively) locks the whole data structure before making any changes. Thus, a thread executing `reorganize` does not have to worry about the interference from other threads.

Figure 8.8 shows the proof outline. The first three lines provide the specification of the implementation-specific helper function `reorganizeContents` that we assume in this proof. The

precondition requires access to the physical resources of the root and disk node. The postcondition then ensures that root node is empty and the disk node contains the full logical contents. Note that we here use the abbreviation $\lambda_\bot := \lambda k.\, \bot$.

The linearization point is the call to `reorganizeContents`. The proof proceeds in a similar fashion as the previous proofs. We discuss only the aspects relevant to maintaining the invariant $\boxed{\mathsf{mcs\_inv}(r, d)}$. First note that for any $C : \mathsf{KS} \to \mathbb{N}_\bot$ we have $\bar{C} = C$. Hence, the two conjuncts $C'_r = \lambda_\bot$ and $\mathsf{cir}(C'_d, C_r, C_d)$ together with $\mathsf{cir}(H, C_r, C_d)$ from the invariant before the call to `reorganizeContents` trivially imply $\mathsf{cir}(H, C'_r, C'_d)$.

As we also need to update the ghost resource $\boxed{\bullet\, C_d(k)}^{\gamma_{d(k)}}$ to $\boxed{\bullet\, C'_d(k)}^{\gamma_{d(k)}}$ for every key k to preserve the last conjunct of the invariant, we need to show $C_d(k) \leq C'_d(k)$. To see this, observe that we have:

$$\begin{aligned}
\mathsf{cir}(H, \lambda_\bot, C'_d) &\Leftrightarrow C'_d = \bar{H} \\
&\Rightarrow C'_d(k) = \max\{t \mid (k,t) \in H\} \\
&\Leftrightarrow C'_d(k) = \max\{t \mid (k,t) \in H \cup C_d\} \\
&\Rightarrow C'_d(k) \geqslant \max\{t \mid (k,t) \in C_d\} \\
&\Leftrightarrow C'_d(k) \geqslant C_d(k)
\end{aligned}$$

The second last step follows from $C_d \subseteq H$, which is implied by $\boxed{\circ\, C_d}^{\gamma_s}$ in the proof context and $\boxed{\bullet\, H}^{\gamma_s}$ from the invariant before the linearization point. The preservation of the remaining parts of the invariant follow easily.

1 $\{\mathsf{node}_n(n, C_n)\}$ `inContents n k` $\{v.\, \mathsf{node}_n(n, C_n) * C_n(k) = v\}$
2 $\boxed{\mathsf{mcs\_inv}(r, d)} \mathrel{-\!\!*} n \in \{r, d\} \mathrel{-\!\!*} \langle\, \mathsf{True} \,\rangle$ `lockNode n` $\langle\, \mathsf{N}(r, n, C_n) \,\rangle$
3 $\boxed{\mathsf{mcs\_inv}(r, d)} \mathrel{-\!\!*} n \in \{r, d\} \mathrel{-\!\!*} \mathsf{N}(r, n, C_n) \mathrel{-\!\!*} \langle\, \mathsf{True} \,\rangle$ `unlockNode n` $\langle\, \mathsf{True} \,\rangle$
4
5 $\{\boxed{\mathsf{mcs\_inv}(r, d)} * \mathsf{mcs\_sr}(k, t_0)\} * \langle\, t\ H.\ \mathsf{MCS}(t, H) \,\rangle$
6 **let** search r d k =
7 lockNode r;
8 $\{\mathsf{N}(r, C_r) * \boxed{\circ\ H_r}^{\gamma_s} * (C_r \subseteq H_r)\}$
9 **let** t' = inContents r k **in**
10 $\{\mathsf{N}(r, r, C_r) * C_r(k) = t'\}$
11 **if** t' != $\bot$ **then begin**
12 $\{\mathsf{N}(r, r, C_r) * C_r(k) = t' \neq \bot\}$
13 (* Linearization point *)
14 $\{\mathsf{N}(r, r, C_r) * C_r(k) = t' * \mathsf{MCS}(r, t, H) * \bar{H}(k) = C_r(k) * t_0 \leq \bar{H}(k)\}$
15 $\{\mathsf{N}(r, r, C_r) * \mathsf{MCS}(r, t, H) * (k, t) \in H * t_0 \leq t'\}$
16 unlockNode r; t'
17 $\langle\, t'.\ \mathsf{MCS}(r, t, H) * (k, t') \in H * t_0 \leq t' \,\rangle$
18 **end**
19 **else begin**
20 $\{\mathsf{N}(r, r, C_r) * C_r(k) = \bot\}$
21 $\{\mathsf{N}(r, r, C_r) * t_0 \leq \bar{H}(k) * \bar{H}(k) = C_d(k)\}$
22 $\{\mathsf{N}(r, r, C_r) * t_0 \leq t_1 * \boxed{\circ\ t_1}^{\gamma_{d(k)}}\}$
23 unlockNode r;
24 $\{t_0 \leq t_1 * \boxed{\circ\ t_1}^{\gamma_{d(k)}}\}$
25 lockNode d;
26 $\{\mathsf{N}(r, d, C_d) * t_0 \leq t_1 * \boxed{\circ\ t_1}^{\gamma_{d(k)}}\}$
27 $\{\mathsf{N}(r, d, C_d) * t_0 \leq C_d(k)\}$
28 **let** t' = inContents d k **in**
29 $\{\mathsf{N}(r, d, C_d) * t_0 \leq C_d(k) * C_d(k) = t'\}$
30 (* Linearization point *)
31 $\{\mathsf{N}(r, d, C_d) * \mathsf{MCS}(r, t, H) * t_0 \leq C_d(k) * C_d(k) = t'\}$
32 $\{\mathsf{N}(r, d, C_d) * \mathsf{MCS}(r, t, H) * (k, t') \in H * t_0 \leq t'\}$
33 unlockNode d; t'
34 $\langle\, t'.\ \mathsf{MCS}(r, t, H) * (k, t') \in H * t_0 \leq t' \,\rangle$
35 **end**
36 $\langle\, t'.\ \mathsf{MCS}(r, t, H) * t_0 \leq t' * (k, t') \in H \,\rangle$

Figure 8.5: Proof of `search` for the two-node multicopy template

$$
\begin{aligned}
\mathsf{mcs\_inv}(r,d) \coloneqq \exists\, t\, H\, C_r\, C_d.\ & \\
& \mathsf{MCS}^\bullet(r,t,H) \\
& * \boxed{\bullet\ H}^{\gamma_s} \\
& * \boxed{\tfrac{1}{2}\ \bullet\ t}^{\gamma_t} \\
& * (\exists b_r.\ \mathsf{lk}(r) \mapsto b_r * (b_r\ ?\ \mathsf{True} : \mathsf{N}(r,r,C_r))) * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} \\
& * (\exists b_d.\ \mathsf{lk}(d) \mapsto b_d * (b_d\ ?\ \mathsf{True} : \mathsf{N}(r,d,C_r))) * \boxed{\tfrac{1}{2} C_d}^{\gamma_{c(d)}} \\
& * \mathsf{cir}(H, C_r, C_d) \\
& * \mathsf{maxTS}(t, H) \\
& * \mathop{\Large *}_{k \in \mathsf{KS}} \boxed{\bullet\ C_d(k)}^{\gamma_{d(k)}}
\end{aligned}
$$

where $\mathsf{N}(r,n,C_n) \coloneqq \mathsf{node}(r,n,C_n) * \boxed{\circ\ C_n}^{\gamma_s} * \boxed{\tfrac{1}{2} C_n}^{\gamma_{c(n)}} * \left(r = n\ ?\ \boxed{\tfrac{1}{2}\ \bullet\ t}^{\gamma_t} : \mathsf{True}\right)$

$\mathsf{cir}(H, C_r, C_d) \coloneqq \bar{H} = \lambda k.\ (C_r(k) \neq \bot\ ?\ C_r(k) : C_d(k))$

$\mathsf{maxTS}(H,t) \coloneqq \forall\, k, t'.\ (k,t') \in H \to t' < t$

Figure 8.6: The full invariant for the two-node multicopy template. Changes to the invariant given in Fig. 8.4 are highlighted in red.

$\{\mathsf{node}(r, r, C_r)\}$ `addContents r k t` $\{v.\, \mathsf{node}(r, r, C'_r) * C'_r = (v\ ?\ C_r[k \rightarrowtail t] : C_r)\}$
$\left\{\boxed{q \bullet t}^{\gamma_t} * q > 0\right\}$ `readClock ()` $\left\{v. \boxed{q \bullet t}^{\gamma_t} * v = t\right\}$
$\left\{\boxed{\bullet\ t}^{\gamma_t}\right\}$ `incrementClock ()` $\left\{\boxed{\bullet\ t+1}^{\gamma_t}\right\}$

$\left\{\boxed{\mathsf{mcs\_inv}(r, d)}\right\} * \langle\, t\, H.\, \mathsf{MCS}(t, H)\,\rangle$
let upsert r k =
  lockNode r;
  $\{\mathsf{N}(r, r, C_r)\}$
  $\left\{\mathsf{node}(r, r, C_r) * \boxed{\circ\ C_r}^{\gamma_s} * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2} \bullet t_1}^{\gamma_t}\right\}$
  let t = readClock () **in**
  $\left\{\mathsf{node}(r, r, C_r) * \boxed{\circ\ C_r}^{\gamma_s} * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2} \bullet t_1}^{\gamma_t} * t = t_1\right\}$
  let res = addContents r k t **in**
  if res **then begin**
    $\left\{\mathsf{node}(r, r, C'_r) * \boxed{\circ\ C_r}^{\gamma_s} * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2} \bullet t}^{\gamma_t} * C'_r = C_r[k \rightarrowtail t]\right\}$
    (* Linearization point *)
    $\left\{\mathsf{node}(r, r, C'_r) * \boxed{\circ\ C_r}^{\gamma_s} * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2} \bullet t}^{\gamma_t} * C'_r = C_r[k \rightarrowtail t] * \mathsf{MCS}(r, t', H')\right\}$
    $\left\{\mathsf{node}(r, r, C'_r) * \boxed{\circ\ C_r}^{\gamma_s} * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2} \bullet t}^{\gamma_t} * C'_r = C_r[k \rightarrowtail t] * \mathsf{MCS}(r, t, H)\right\}$
    incrementClock ();
    $\left\{\begin{array}{l}\mathsf{node}(r, r, C'_r) * \boxed{\circ\ C_r}^{\gamma_s} * \boxed{\tfrac{1}{2} C'_r}^{\gamma_{c(r)}} * C'_r = C_r[k \rightarrowtail t] \\ * \boxed{\tfrac{1}{2} \bullet t+1}^{\gamma_t} * H' = H \cup \{(k, t)\} * \mathsf{MCS}(r, t+1, H')\end{array}\right\}$
    $\{\mathsf{N}(r, r, C'_r)\} * \langle\, \mathsf{MCS}(r, t+1, H \cup \{(k, t)\})\,\rangle$
    unlockNode r
    $\langle\, \mathsf{MCS}(r, t+1, H \cup \{(k, t)\})\,\rangle$
  end
  else begin
    $\left\{\mathsf{node}(r, r, C'_r) * \boxed{\circ\ C_r}^{\gamma_s} * \boxed{\tfrac{1}{2} C_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2} \bullet t}^{\gamma_t} * C'_r = C_r\right\}$
    $\{\mathsf{N}(r, r, C_r)\}$
    unlockNode r;
    upsert r k
  end
$\langle\, \mathsf{MCS}(t+1, H \cup \{(k, t)\})\,\rangle$

Figure 8.7: Proof of `upsert` for the two-node multicopy template

$\{\mathsf{node}(r, r, C_r) * \mathsf{node}(r, d, C_d)\}$
`reorganizeContents r d`
$\{\mathsf{node}(r, r, C'_r) * \mathsf{node}(r, d, C'_d) * C'_r = \lambda_\perp * \mathsf{cir}(C'_d, C_r, C_d)\}$

$\{\boxed{\mathsf{mcs\_inv}(r, d)}\}$ * $\langle\, t\, H.\ \mathsf{MCS}(r, t, H)\,\rangle$
let `reorganize r d =`
  `lockNode r;`
  $\{\mathsf{N}(r, r, C_r)\}$
  `lockNode d;`
  $\{\mathsf{N}(r, r, C_r) * \mathsf{N}(r, d, C_d)\}$
  `(* Linearization point *)`
  $\{\mathsf{N}(r, r, C_r) * \mathsf{N}(r, d, C_d) * \mathsf{MCS}(r, t, H)\}$
  $\left\{\begin{array}{r} \mathsf{node}(r, r, C_r) * \boxed{\circ\, C_r}^{\gamma_s} * \boxed{\tfrac{1}{2}C_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2}\bullet t}^{\gamma_t} \\ * \mathsf{node}(r, d, C_d) * \boxed{\circ\, C_d}^{\gamma_s} * \boxed{\tfrac{1}{2}C_d}^{\gamma_{c(d)}} * \mathsf{MCS}(r, t, H) \end{array}\right\}$
  `reorganizeContents r d;`
  $\left\{\begin{array}{r} \mathsf{node}(r, r, C'_r) * \boxed{\circ\, C_r}^{\gamma_s} * \boxed{\tfrac{1}{2}C'_r}^{\gamma_{c(r)}} * \boxed{\tfrac{1}{2}\bullet t}^{\gamma_t} * C'_r = \lambda_\perp \\ * \mathsf{node}(r, d, C'_d) * \boxed{\circ\, C_d}^{\gamma_s} * \boxed{\tfrac{1}{2}C'_d}^{\gamma_{c(d)}} * \mathsf{MCS}(r, t, H) \end{array}\right\}$
  $\{\mathsf{N}(r, r, C'_r) * \mathsf{N}(r, d, C'_d)\}$ * $\langle\, \mathsf{MCS}(r, t, H)\,\rangle$
  `unlockNode r;`
  `unlockNode d`
$\langle\, \mathsf{MCS}(r, t, H)\,\rangle$

Figure 8.8: Proof of maintenance operation for the two-node multicopy template.

CHAPTER 9

Verifying a General Multicopy Template

In this chapter, we present a general template for multicopy structures that comprise an unbounded number of disk nodes. That is, the template generalizes the LSM (log-structured merge) tree discussed in §7.3. Again, we prove linearizability of the template by verifying that all operations satisfy the atomic specifications assumed by Theorem 7.1.

The high-level proof closely follows that of the two-node template and relies on the same high-level invariants. In particular, we build again on the notion of the contents-in-reach of a node, which captures the relationship between the logical contents of the multicopy structure and the physical contents of individual nodes in its graph. However, there is one new technical challenge that we have to solve here: due to the unbounded number of disk nodes, the definition of the contents-in-reach now involves a computation over a graph of unbounded size. We will use the flow framework (Chapter 5) to define the contents-in-reach in terms of a suitable flow, enabling us to carry out all proofs using only local reasoning about bounded graph regions.

9.1 THE GENERAL MULTICOPY TEMPLATE

We split the template into two parts. The first part is a template for `search` and `upsert` that works on general multicopy structures, i.e., arbitrary DAGs with locally disjoint edgesets. The second part is a template for a maintenance operation that generalizes the compaction mechanism found in existing list-based LSM tree implementations to tree-like multicopy structures.

Multicopy operations. Figure 9.1 shows the code of the template for the generic multicopy operations. The operations `search` and `upsert` closely follow the high-level description of these operations on the LSM tree (Section 7.3). As with previous templates, the operations are defined in terms of implementation-specific helper functions. These functions are similar to those introduced for the two-node multicopy template. Only `findNext` is new. It returns the unique successor of the given node n and query key k (i.e., the node n' satisfying $k \in \mathsf{es}(n, n')$) if such a successor exists.

The `search` operation calls the recursive function `traverse` on the root node and generalizes the `search` operation for the two-node template to an unbounded number of disk nodes. Note that in contrast to lock-coupling, the thread executing the `search` holds no locks at the points when `traverse` is called recursively. The `upsert` operation is identical to the one on the two-node template.

```
let rec traverse n k =
  lockNode n;
  let t = inContents n k in
  if t != ⊥ then begin
    unlockNode n; t
  end
  else begin
    match findNext n k with
    | Some n' ->
      unlockNode n;
      traverse n' k
      end
    | None -> unlockNode n; ⊥
  end

let search r k = traverse r k
```

```
let rec upsert r k =
  lockNode r;
  let t = readClock () in
  let res = addContents r k t in
  if res then begin
    incrementClock ();
    unlockNode r
  end
  else begin
    unlockNode r;
    upsert r k
  end
```

Figure 9.1: The general template for multicopy operations `search` and `upsert`. The template can be instantiated by providing implementations of helper functions `inContents`, `findNext`, and `addContents`. `inContents` $n\,k$ returns $C_n(k)$. `findNext` $n\,k$ returns `Some` n' if n' is the unique node such that $k \in$ es(n, n'), or `None` otherwise. `addContents` $r\,k$ adds the key k with the current timestamp t to the contents of r. The return value of `addContents` is a boolean which indicates whether the insertion was successful (e.g. if r is full, insertion may fail leaving r's contents unchanged).

```
let rec compact n =
  lockNode n;
  if atCapacity n then begin
    let m = chooseNext n in
    lockNode m;
    mergeContents n m;
    unlockNode n;
    unlockNode m;
    compact m
  end
  else
    unlock n
```

Figure 9.2: Maintenance template for tree-like multicopy structures. The template can be instantiated by providing implementations of helper functions `atCapacity`, `chooseNext`, and `mergeContents`. `atCapacity` n returns a boolean value indicating whether node n has reached its capacity. The helper function `chooseNext` n returns a successor m of n into which n should be compacted. Note that m may be freshly allocated. Finally, `mergeContents` $n\,m$ (partially) merges the contents of n into m.

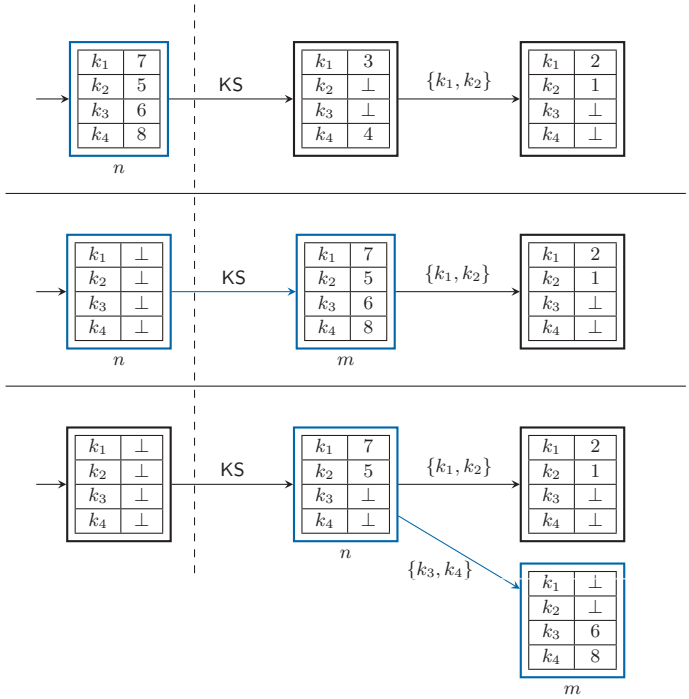

Figure 9.3: Possible execution of the `compact` operation shown in Fig. 9.2. Edges are labeled with their edgesets. The nodes n and m in each iteration are marked in blue.

Maintenance operation. For the maintenance template, we consider a generalized version of the compaction operation found in LSM tree implementations such as LevelDB [Google, 2020] and Apache Cassandra [Apache Software Foundation, 2020a, Jonathan Ellis, 2011]. While those implementations work on lists for the top-level multicopy structure, our generalized template supports arbitrary tree-like multicopy structures. The code is shown in Fig. 9.2. The template uses the helper function `atCapacity` n to test whether the size of n (i.e., the number of non-$\perp$ entries in n's contents) exceeds an implementation-specific threshold. If not, then the operation simply terminates. In case n is at capacity, the function `chooseNext` is used to pick a successor node m of n to merge some of the contents of n into m. The merge is done by the helper function `mergeContents`. It must ensure that all merged keys k satisfy $k \in \mathsf{es}(n, m)$. The node m returned by `chooseNext` may be freshly allocated in which case the new edgeset $\mathsf{es}(n, m)$ must be disjoint from all edgesets for the other successors m' of n. Note that the maintenance template never removes nodes from the structure. In practice, the depth of the structure is bounded by letting the capacity of nodes grow exponentially with the depth. Figure 9.3 shows the intermediate states of a potential execution of the `compact` operation.

9.2 CORRECTNESS PROOF FOR THE GENERAL MULTICOPY TEMPLATE

We will show that operations of the template satisfy the atomic specifications required by Theorem 7.1 from Chapter 7:

$$\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \mathsf{mcs\_sr}(k, t_0) \mathrel{-\!\!*} \left\langle\, t\, H.\ \mathsf{MCS}(r,t,H) \,\right\rangle \texttt{search r k} \left\langle\, t'.\ \mathsf{MCS}(r,t,H) * (k,t') \in H * t_0 \leq t' \,\right\rangle$$

$$\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \left\langle\, t\, H.\ \mathsf{MCS}(r,t,H) \,\right\rangle \texttt{upsert r k} \left\langle\, \mathsf{MCS}(r,t+1,H \cup (k,t)) \,\right\rangle$$

$$\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \left\langle\, t\, H.\ \mathsf{MCS}(r,t,H) \,\right\rangle \texttt{flush r} \left\langle\, \mathsf{MCS}(r,t,H) \,\right\rangle$$

$$\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \left\langle\, t\, H.\ \mathsf{MCS}(r,t,H) \,\right\rangle \texttt{compact r} \left\langle\, \mathsf{MCS}(r,t,H) \,\right\rangle$$

Here, $\mathsf{mcs\_inv}(r)$ is the invariant guarding the shared state of the general multicopy structure. We next discuss each operation in detail and prove its corresponding specification, starting with the search operation.

9.2.1 PROVING SEARCH RECENCY

We argue that the high-level proof of search recency for the two-node multicopy template discussed in Section 8.2.1 generalizes to the new template. In particular, we can reuse the same high-level invariants that we identified in the earlier proof. Recall the notion of contents-in-reach $C_{ir}(n)\colon \mathsf{KS} \to \mathbb{N}_\perp$ of a node n which was critical for that proof. The value $C_{ir}(n)(k)$ is the first copy of key k in the data structure that a search starting at node n will find. Invariant 1 then stated the property that the logical contents $\bar{H}$ of a multicopy structure is the contents-in-reach of its root node. This invariant captures the intuition that a search that starts at the root and that is not interfered with by concurrent threads will find the correct copy of its query key. Invariants 2 and 3 then provide the necessary auxiliary properties to prove search recency in the presence of interference. Invariant 2 states that the contents-in-reach of every node can only increase for every key over time. Finally, Invariant 3 captures the fact that all copies of keys in the node-level contents have been upserted at some point in the past.

In addition to these three general invariants that every multicopy structure must satisfy, we need an inductive invariant for the traversal performed by the search: we require as a precondition for `traverse n k` that $C_{ir}(n)(k) \geqslant t_0$ where t_0 is the logical timestamp of k at the point when `search` was invoked. To see that this property holds initially for the call `traverse r k` in `search`, let $\bar{H}_0$ be the logical contents at the time point when `search` was invoked. By assumption we have $\bar{H}_0(k) = t_0$. Moreover, Invariant 1 implies that we must have had $C_{ir}(r)(k) = t_0$ at this point. Since $C_{ir}(r)(k)$ only increases over time according to Invariant 2, we can conclude that $C_{ir}(r)(k) \geqslant t_0$ when `traverse` is called. We next show that the traversal invariant is maintained by `traverse` and is sufficient to prove search recency.

Consider a call `traverse n k` such that $C_{ir}(n)(k) \geqslant t_0$ holds initially. We must show that the call returns t such that $t \geqslant t_0$ and $(k,t) \in H$. We know that the call to `inContents` on line 3 returns

t such that $t = C_n(k)$. Let us first consider the case where $t \neq \bot$. In this case, `traverse` returns t on line 5. By definition of $C_{ir}(n)$ we have $C_{ir}(n)(k) = C_n(k)$. Hence, the precondition $C_{ir}(n)(k) \geqslant t_0$ together with Invariant 2 guarantee $t \geqslant t_0$. Moreover, Invariant 3 guarantees $(k, t) \in H$.

Now consider the case where $C_n(k) = t = \bot$, indicating that no copy has been found for k in n. In this case, `traverse` calls `findNext` to obtain the successor node of n and k. In the case where the successor n' exists (line 9), we know that $k \in \mathsf{es}(n, n')$ must hold. Hence, by definition of contents-in-reach we must have $C_{ir}(n)(k) = C_{ir}(n')(k)$. From $C_{ir}(n)(k) \geqslant t_0$ and Invariant 2, we can then conclude $C_{ir}(n')(k) \geqslant t_0$, i.e. the precondition for the recursive call to `traverse` on line 11 is satisfied and search recency follows by induction.

On the other hand, if n does not have any next node, then `traverse` returns $\bot$ (line 13), indicating that k has not yet been upserted at all so far. In this case, by definition of contents-in-reach we must have $C_{ir}(n)(k) = \bot$. Invariant 2 then guarantees $\bot \geqslant t_0$ which in turn implies $t_0 = \bot$. Hence, search recency holds trivially in this case.

In the remainder of this section, we show how to formalize the above proof in Iris. The key technical challenge we will have to solve here, compared to the proof of search recency for the two-node template, is that the contents-in-reach is now a recursive function defined over an unbounded graph. This makes it more difficult to obtain a simple local proof that involves reasoning only about a bounded number of nodes in the graph at a time. We will solve this challenge using the flow framework, similarly to how we dealt with the inset function in the case of the single-copy templates.

Encoding contents-in-reach using flows. Let us revisit the recursive definition of contents-in-reach given in Equation (8.1). The computation of contents-in-reach proceeds bottom-up in the multicopy structure graph starting from the leaves and continues recursively towards the root node. That is, the computation proceeds *backwards* with respect to the direction of the graph's edges. This makes a direct encoding of contents-in-reach in terms of flows difficult because the flow equation describes computations that proceed in forward direction. We side-step this problem by following an approach where we track the purported contents-in-reach of every node as explicit ghost state in the invariant and then use a flow to propagate these values forward in the graph so as to check that they are indeed consistent with the recursive definition (8.1).

In the following, we denote by B_n the purported value of $C_{ir}(n)$ for every node n in the graph. We will track this value in an auxiliary ghost location associated with each node in the invariant $\mathsf{mcs\_inv}(r)$. The flow of a node n in our encoding then captures the assumptions that the $B_{n'}$ make about $C_{ir}(n)$ for all predecessor nodes n' of n. The node-level invariant for n enforced by $\mathsf{mcs\_inv}(r)$ will ensure that these assumptions are consistent with the purported value B_n for $C_{ir}(n)$ at n.

The cancellative monoid M for the flow domain of this encoding consists of multisets of key-timestamp pairs $M := \mathsf{KS} \times \mathbb{N}_\bot \to \mathbb{N}$. The edge function for the flow graph induced by the multicopy structure is defined as follows:

$$e(n, n')(\_) := \chi(\{(k, t) \mid k \in \mathsf{es}(n, n') \land C_n(k) = \bot \land B_n(k) = t\}) \tag{9.1}$$

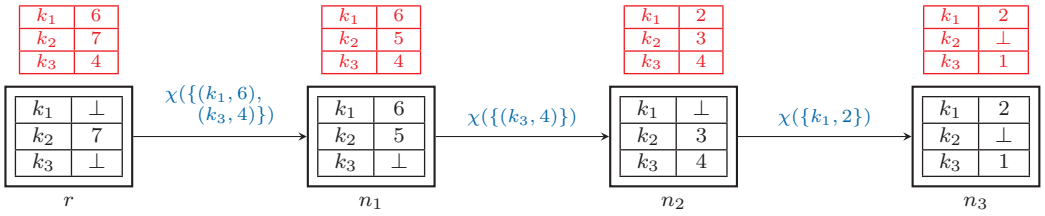

Figure 9.4: Flow encoding of contents-in-reach for multicopy structure shown in Fig. 7.6.

Here, χ takes a set to its characteristic function. Note that this definition is independent of the flow of node n. That is, the outflow of n via edge (n, n') depends only on C_n, B_n and $\mathsf{es}(n, n')$. This outflow is the multiset of all pairs (k, t) for which we must guarantee $C_{ir}(n')(k) = t$ in order to obtain $B_n = C_{ir}(n)$. Finally, we let the global inflow of the flow graph assign the empty multiset to every node.

Figure 9.4 illustrates the encoding for the multicopy list structure depicted in Fig. 7.6. Each node n is labeled with its contents C_n (in black). Above each node we show B_n (in red). Each edge is labeled in blue with the outflow going from the source to the sink node of the edge. As every node has at most one predecessor, the outflow labeling each edge is also the flow of the sink node of that edge.

Let I_n be the node-level flow interface of node n in this flow graph (i.e., the inflow $I_n.in(n)$ coincides with n's flow). Let further $\mathsf{outs}(I_n)$ be the set of all keys that are in the outflow of I_n, i.e.

$$\mathsf{outs}(I_n) \coloneqq \{k \mid \exists n'\, t.\, I_n.out(n')(k,t) > 0\} \quad .$$

In order to ensure that we indeed must have $B_n = C_{ir}(n)$ for all nodes n, the invariant $\mathsf{mcs\_inv}(r)$ needs to enforce the following node-local conditions:

$$\phi_1(n, B_n, C_n, I_n) \coloneqq \forall k.\, k \in \mathsf{outs}(I_n) \vee B_n(k) = C_n(k) \tag{9.2}$$
$$\phi_2(n, B_n, I_n) \coloneqq \forall k\, t.\, I_n.in(n)(k,t) > 0 \Rightarrow B_n(k) = t \tag{9.3}$$

The following lemma states the correctness of this encoding.

Lemma 9.1 *If $\phi_1(n, B_n, C_n, I_n)$ and $\phi_2(n, B_n, I_n)$ hold for all nodes n in the flow graph obtained from a multicopy structure as described above, then $B_n = C_{ir}(n)$.*

Proof. The proof proceeds by induction over the inverse topological order of the multicopy structure graph. Let k be a key and n a node in the graph. We show $B_{n'}(k) = C_{ir}(n')(k)$, provided $B_{n'} = C_{ir}(n')$ holds for all successors n' of n in the topological order. We have to consider three cases according to the definition of $C_{ir}(n)$ (Equation (8.1).

First, suppose $C_n(k) \neq \bot$. In this case, we have $C_{ir}(n)(k) = C_n(k)$. Moreover, Equation (9.1) implies $k \notin \mathsf{outs}(I_n)$. Hence, we can conclude from $\phi_1(n, B_n, C_n, I_n)$ that $B_n(k) = C_n(k)$ holds, as well.

Next, suppose $C_n(k) = \bot$ and $k \in \mathsf{es}(n, n')$ for some n'. Then Equation (9.1) implies $I_n.out(n')(k, B_n(k)) > 0$. Since we have a valid flow graph, the interfaces I_n and $I_{n'}$ must compose, which implies $I_{n'}.in(n')(k, B_n(k)) > 0$. It follows from $\phi_2(n', B_{n'}, I_{n'})$ that we then must have $B_{n'}(k) = B_n(k)$. By induction hypothesis we conclude $B_{n'}(k) = C_{ir}(n')(k)$ and, hence, $B_n(k) = C_{ir}(n')(k) = C_{ir}(n)(k)$ by Equation (8.1).

Finally, consider the case where $C_n(k) = \bot$ and for all nodes n', $k \notin \mathsf{es}(n, n')$. In this case, Equation (9.1) again implies $k \notin \mathsf{outs}(I_n)$ and $B_n(k) = C_n(k) = \bot$ follows from $\phi_1(n, B_n, C_n, I_n)$. Thus, we again conclude $B_n(k) = \bot = C_{ir}(n)(k)$ by Equation (8.1). □

Note that all nodes in the flow graph shown in Fig. 9.4 satisfy these conditions and, indeed, the annotated B_n are the correct contents-in-reach for this multicopy structure.

We next discuss how to incorporate this encoding into a general multicopy structure invariant $\mathsf{mcs\_inv}(r)$.

Invariant for multicopy data structure. The invariant $\mathsf{mcs\_inv}(r)$ for the general multicopy structure is shown in Fig. 9.5. It generalizes the invariant for the two-node multicopy template in Fig. 8.6 by incorporating many of the ideas that we previously introduced for the proofs of the single-copy templates (Chapter 6). For presentation purposes, we omit some details of the invariant for now that we will only need when discussing the correctness of the maintenance operation. We discuss each part of Fig. 9.5 in detail:

- The existentially quantified variables t, H and I are to be interpreted as the current clock value, the current upsert history, and the current global interface of the graph, respectively.
- Similar to the two-node template, we use two abstract predicates $\mathsf{MCS}^\bullet(r, t, H)$ and $\mathsf{MCS}(r, t, H)$ so that the invariant and the atomic triples of the data structure operations can share a consistent view of the abstract state of the multicopy structure.
- We use appropriate authoritative RAs at ghost locations γ_s and γ_t, to keep track of the upsert history H and the global clock t. As for the two-node template, we split ownership of the global clock between the invariant and the node predicate for the root node to capture Invariant 5. We will need this property again for the proof of `upsert` later.
- We use an authoritative RA of flow interfaces at ghost location γ_I to keep track of the global interface I, which is composed of singleton interfaces I_n for each node $n \in \mathsf{dom}(I)$. The I_n are tied to the implementation-specific physical representation of the individual nodes via the predicates N and S as explained below.
- The ghost resource $\boxed{\bullet\, \mathsf{dom}(I)}^{\gamma_f}$ keeps track of the footprint of the data structure using the authoritative RA of sets of nodes. This resource plays a similar role in the proof as in the proofs

$$
\begin{aligned}
\mathsf{mcs\_inv}(r) :=\ & \exists\, t\, H\, I. \\
& \quad \mathsf{MCS}^{\bullet}(r,t,H) \\
& * \boxed{\bullet\ H}^{\gamma_s} * \boxed{\tfrac{1}{2}\bullet\ t}^{\gamma_t} * \boxed{\bullet\ I}^{\gamma_I} * \boxed{\bullet\ \mathsf{dom}(I)}^{\gamma_f} \\
& * I.in = \lambda_0 \\
& * \mathsf{inFP}(r) \\
& * \mathsf{maxTS}(t,H) \\
& * \mathop{\Large *}_{n\in\mathsf{dom}(I)} \exists b_n\, C_n\, B_n.\ \ \mathsf{lk}(n) \mapsto b_n \\
& \qquad\qquad\qquad\qquad * (b_n\ ?\ \mathsf{True} : \mathsf{N}(r,n,C_n,B_n)) \\
& \qquad\qquad\qquad\qquad * \mathsf{S}(r,n,C_n,B_n,H)
\end{aligned}
$$

where $\mathsf{maxTS}(H,t) := \forall\, k,t'.\ (k,t') \in H \to t' < t$

$$
\begin{aligned}
\mathsf{N}(r,n,C_n,B_n) :=\ & \exists es,t.\ \mathsf{node}(r,n,es,C_n) \\
& * \boxed{\tfrac{1}{2}es}^{\gamma_{e(n)}} * \boxed{\tfrac{1}{2}C_n}^{\gamma_{c(n)}} * \boxed{\tfrac{1}{2}B_n}^{\gamma_{b(n)}} * \boxed{\circ\ C_n}^{\gamma_s} \\
& * \Big(n = r\ ?\ \boxed{\tfrac{1}{2}\bullet\ t}^{\gamma_t} : \mathsf{True}\Big)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{S}(r,n,C_n,B_n,H) :=\ & \exists es, I_n. \\
& * \boxed{\tfrac{1}{2}es}^{\gamma_{e(n)}} * \boxed{\tfrac{1}{2}C_n}^{\gamma_{c(n)}} * \boxed{\tfrac{1}{2}B_n}^{\gamma_{b(n)}} \\
& * \boxed{\circ I_n}^{\gamma_I} * \mathsf{dom}(I_n) = \{n\} * \mathsf{inFP}(n) * \mathsf{closed}(es) \\
& * I_n.out = \lambda n'.\ \chi(\{(k,t) \mid k \in es(n') \wedge C_n(k) = \bot \wedge B_n(k) = t\}) \\
& * \big(n = r\ ?\ B_n = \bar{H} * I_n.in = \lambda_0 : \mathsf{True}\big) \\
& * \mathop{\Large *}_{k\in\mathsf{KS}} \boxed{\bullet\ B_n(k)}^{\gamma_{cir(n)(k)}} \\
& * \phi_1(n,B_n,C_n,I_n) * \phi_2(n,B_n,I_n)
\end{aligned}
$$

$$
\mathsf{inFP}(n) := \boxed{\circ\ \{n\}}^{\gamma_f}
$$

$$
\mathsf{closed}(es) := \forall n'.\ es(n') \neq \emptyset \to \mathsf{inFP}(n')
$$

Figure 9.5: The invariant for the general multicopy template.

of the single-copy structures discussed earlier. Notably we use it to maintain the invariant of `traverse` that the currently visited node n remains part of the data structure while n is unlocked. We require that the global interface I has no inflow ($I.in = \lambda_0$).

- The condition $\mathsf{inFP}(r)$ guarantees that r is always in the domain of the data structure.

- The condition $\mathsf{maxTS}(H, t)$ again captures Invariant 4, which will be needed for the proof of `upsert`.

- The resources for every node n are split between the two predicates $\mathsf{N}(r, n, C_n, B_n)$ and $\mathsf{S}(r, n, C_n, B_n, H)$. The latter is always owned by the invariant whereas the former is protected by n's lock and transferred between the invariant and the thread's local state upon locking the node and vice versa upon unlocking, as usual. We next discuss these two predicates in detail.

- The first conjunct of $\mathsf{N}(r, n, C_n, B_n)$ is the implementation-specific node predicate $\mathsf{node}(r, n, es, C_n)$. For each specific implementation of the template, this predicate must tie the physical representation of the node n to its contents C_n and a function $es : \mathfrak{N} \to \wp(\mathsf{KS})$ which captures the edgesets of n's outgoing edges. We require that node is not duplicable:

$$\mathsf{node}(r, n, es, C_n) * \mathsf{node}(r', n, es', C_n') \vdash \mathsf{False}$$

  Moreover, node must guarantee that the edgesets are disjoint:

$$\mathsf{node}(r, n, es, C_n) \vdash \forall n_1\, n_2.\, n_1 = n_2 \vee es(n_1) \cap es(n_2) = \emptyset$$

  We include r in the parameters of node so that the implementation can choose different representations for memory and disk nodes.

- The fractional resources at ghost locations $\gamma_{e(n)}$, $\gamma_{c(n)}$, and $\gamma_{b(n)}$ ensure that the predicates N and S agree on es, C_n, and B_n even when n is locked.

- The ghost resource $\boxed{\circ\, C_n}^{\gamma_s}$ when combined with $\boxed{\bullet\, H}^{\gamma_s}$ implies $C_n \subseteq H$, which captures Invariant 3.

- As discussed previously, the final conjunct of N guarantees sole ownership of the global clock by a thread holding the lock on the root node.

- Moving on to $\mathsf{S}(r, n, C_n, B_n, H)$, this predicate contains those resources of n that are available to all threads at all times. In particular, the resource $\boxed{\circ I_n}^{\gamma_I}$ guarantees that all the singleton interfaces I_n compose to the global interface I, thus, satisfying the flow equation. Similarly, the predicate $\mathsf{inFP}(n)$ guarantees that n remains in the data structure at all times. The predicate $\mathsf{closed}(es)$ ensures that the outgoing edges of n point to nodes which are again in the data structure. Together with the condition $\mathsf{inFP}(r)$, this guarantees that all nodes reachable from r must be in $\mathsf{dom}(I)$.

- The next conjunct of S defines the outflow of the singleton interface I_n according to Equation (9.1) of our flow encoding of contents-in-reach. Note that, even though I_n is not shared with the predicate N, only its inflow can change when n is locked, because the outflow of the interface is determined by C_n, B_n, and es, all of which are protected by n's lock.

1 $\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \mathsf{inFP}(n) \mathrel{-\!\!*} \langle\, \mathsf{True} \,\rangle$ `lockNode n` $\langle\, \mathsf{N}(r, n, I_n, C_n, D_n) \,\rangle$

2 $\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \mathsf{inFP}(n) \mathrel{-\!\!*} \mathsf{N}(r, n, I_n, C_n, D_n) \mathrel{-\!\!*} \langle\, \mathsf{True} \,\rangle$ `unlockNode n` $\langle\, \mathsf{True} \,\rangle$

3

4 $\{\mathsf{node}(r, n, es, C_n)\}$

5 `inContents n k`

6 $\{t.\ \mathsf{node}(r, n, es, C_n) * C_n(k) = t\}$

7

8 $\{\mathsf{node}(r, n, es, C_n)\}$

9 `findNext n k`

10 $\{v.\ \mathsf{node}(r, n, es, C_n) * (v = \mathsf{Some}(n') * k \in es(n') \vee v = \mathsf{None} * \forall n'.\ k \notin es(n'))\}$

Figure 9.6: Specifications of helper functions used by `search`

- The constraint $B_n = \bar{H}$ holds if $r = n$ and implies Invariant 1. We further require here that the interface of the root node has no inflow ($I_n.in = \lambda_0$), a property that we need in order to prove that `upsert` maintains the flow-related invariants. Moreover, similar to our two-node invariant, we use for every key k an authoritative maxnat resource at ghost location $\gamma_{cir(n)(k)}$ to capture Invariant 2.
- The last two conjuncts of S complete our encoding of contents-in-reach and ensure that we must indeed have $B_n = C_{ir}(n)$ at all atomic steps.

Finally, recall from our prior discussion in Section 8.2.1 that we encode the search recency property needed for the meta-level correctness argument (Theorem 7.1) using the predicate $\mathsf{mcs\_sr}(k, t_0)$. This predicate captures the fact that t_0 is the logical timestamp of k at some point in the past. We keep the definition of this predicate the same as for the proof of the two-node multicopy template (cf. Section 8.2.1):

$$\mathsf{mcs\_sr}(k, t_0) \coloneqq \boxed{\circ\ \{(k, t_0)\}}^{\gamma_s} \quad .$$

We now have all the ingredients to proceed with the proof of `search`.

Proof of `search`. We start with the specification of the implementation-specific helper functions assumed by `search` as well as the lock module used in the proof. They are provided in Fig. 9.6. The specifications of `lockNode`, `unlockNode`, and `inContents` are adapted from the ones we assumed for the two-node template algorithm. The precondition $\mathsf{inFP}(n)$ is required for `lockNode` and `unlockNode` in order to ensure that the node n which is operated on belongs to the graph. The function `findNext` assumes ownership of the resources $\mathsf{node}(r, n, I_n, v_n)$ associated with the locked node n and returns a successor node n' of n such that k is in the edgeset of (n, n') if such a node exists.

Figure 9.7 provides the outline of the proof of search recency for the general multicopy template. As most of the actual work is done by the recursive function `traverse`, we start with its atomic

```
1  {mcs_inv(r)} * mcs_sr(k, t0) * ⟨ t H. MCS(r, t, H) ⟩
2  let search r k =
3    {inFP(r) * H̄1(k) = Br(k) * ∘ Br(k)^γcir(r)(k) * t0 ≤ H̄1(k)}
4    {inFP(r) * ∘ t1^γcir(r)(k) * t0 ≤ t1} * ⟨ t H. MCS(r, t, H) ⟩
5    traverse r k
6  ⟨ t'. MCS(r, t, H) * (k, t') ∈ H * t0 ≤ t' ⟩
7
8  {mcs_inv(r)} * inFP(n) * ∘ t1^γcir(n)(k) * t0 ≤ t1 * ⟨ t H. MCS(r, t, H) ⟩
9  let rec traverse n k =
10   lockNode n;
11   {N(r, n, Cn, Bn) * ∘ Bn(k)^γcir(n)(k) * t0 ≤ Bn(k)}
12   let t' = inContents n k in
13   {N(r, n, Cn, Bn) * ∘ Bn(k)^γcir(n)(k) * t0 ≤ Bn(k) * Cn(k) = t'}
14   if t' != ⊥ then begin
15     {N(r, n, Cn, Bn) * t0 ≤ Bn(k) * Cn(k) = t' ≠ ⊥}
16     (* Linearization point *)
17     {N(r, n, Cn, Bn) * t0 ≤ Cn(k) * Cn(k) = t'}
18     {N(r, n, Cn, Bn) * (k, t') ∈ H * t0 ≤ t' * MCS(r, t, H)}
19     unlockNode n; t'
20     ⟨ t'. MCS(r, t, H) * (k, t') ∈ H * t0 ≤ t' ⟩
21   end
22   else begin
23     {N(r, n, Cn, Bn) * t0 ≤ Bn(k) * Cn(k) = t' = ⊥}
24     match findNext n k with
25     | Some n' -> {node(r, n, es, Cn) * ··· * t0 ≤ Bn(k) * Cn(k) = ⊥ * k ∈ es(n')}
26       {node(r, n, es, Cn) * ··· * t0 ≤ Bn(k) * Cn(k) = ⊥ * k ∈ es(n')}
27       {N(r, n, Cn, Bn) * inFP(n') * ∘ t1^γcir(n')(k) * t0 ≤ t1}
28       unlockNode n;
29       {inFP(n') * ∘ t1^γcir(n')(k) * t0 ≤ t1} * ⟨ t H. MCS(r, t, H) ⟩
30       traverse n' k
31       ⟨ t'. MCS(r, t, H) * (k, t') ∈ H * t0 ≤ t' ⟩
32     | None -> {node(r, n, es, Cn) * ··· * t0 ≤ Bn(k) * Cn(k) = ⊥ * ∀n'. k ∉ es(n')}
33       {N(r, n, Cn, Bn) * t0 ≤ Bn(k) * Bn(k) = Cn(k) = ⊥}
34       (* Linearization point *)
35       {N(r, n, Cn, Bn) * t0 = Bn(k) = Cn = ⊥ * MCS(r, t, H)}
36       {N(r, n, Cn, Bn) * (k, ⊥) ∈ H * t0 = ⊥ * MCS(r, t, H)}
37       unlockNode n; ⊥
38   end
39 ⟨ t'. MCS(r, t, H) * (k, t') ∈ H * t0 ≤ t' ⟩
```

Figure 9.7: Proof of `search` for general multicopy template

specification:

$$\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \mathsf{inFP}(n) \mathrel{-\!\!*} \boxed{\circ\, t_1}^{\gamma_{cir(n)(k)}} \mathrel{-\!\!*} t_0 \leq t_1 \mathrel{-\!\!*}$$
$$\langle\, t\ H.\ \mathsf{MCS}(r,t,H) \,\rangle \texttt{traverse n k} \langle\, t'.\ \mathsf{MCS}(r,t,H) * (k,t') \in H * t_0 \leq t' \,\rangle$$

Recall the traversal invariant $t_0 \leq C_{ir}(n)(k)$ that we used in our informal proof of search recency. The ghost resource $\boxed{\circ\, t_1}^{\gamma_{cir(n)(k)}}$, together with $t_0 \leq t_1$ in the precondition of the above specification precisely capture this invariant. In addition, `traverse` assumes the invariant $\mathsf{mcs\_inv}(r)$ and requires that n must be a node in the graph, expressed by the predicate $\mathsf{inFP}(n)$. The operation then guarantees to return t' such that search recency holds.

Let us for now assume that `traverse` satisfies the above specification and focus on the proof of `search`. The precondition of `search r k` assumes the invariant $\mathsf{mcs\_inv}(r)$ and the predicate $\mathsf{mcs\_sr}(k, t_0)$ (line 1). We must, hence, use these to establish the precondition for the call `traverse n k` on line 5. To this end, we open the invariant from which we can directly obtain $\mathsf{inFP}(r)$. Next, we unfold the definition of $\mathsf{mcs\_sr}(k, t_0)$ to obtain $\boxed{\circ\, \{(k, t_0)\}}^{\gamma_s}$. Snapshotting $\boxed{\bullet\, H_1}^{\gamma_s}$ in the invariant for the current upsert history H_1, we can conclude $(k, t_0) \in H_1$ and therefore $t_0 \leq \bar{H}_1(k)$. From $\mathsf{S}(r, r, C_r, B_r)$ in the invariant we can further deduce $\bar{H}_1(k) = B_r(k)$ and $\boxed{\circ\, B_r(k)}^{\gamma_{cir(r)(k)}}$ (line 3). By substituting both $\bar{H}_1(k)$ and $B_r(k)$ with a fresh existentially quantified variable t_1 we obtain the precondition of `traverse` (line 4). We now commit the atomic triple of `search` on the call to `traverse` and immediately obtain the desired postcondition.

Proof of `traverse`. Finally, we prove the assumed specification of `traverse`. The proof starts at line 8. The thread first locks node n which yields ownership of the predicate $\mathsf{N}(r, n, C_n, B_n)$. At this point, we also open the invariant to take a fresh snapshot of the resource $\boxed{\bullet\, B_n(k)}^{\gamma_{cir(n)(k)}}$ to conclude $t_0 \leq B_n(k)$ from the precondition of `traverse` (line 11). Next, the thread executes `inContents n k`. Note that the precondition of for this call, i.e. $\mathsf{node}(r, n, es, C_n)$, is available to us as part of the predicate $\mathsf{N}(r, n, C_n, B_n)$. Depending on the return value t' of `inContents` we end up with two subcases.

In the case where $C_n(k) = t' \neq \bot$ we continue on line 15. The call to `unlockNode` on line 19 will be the linearization point of this case. To obtain the desired postcondition of the atomic triple, we first retrieve $\mathsf{S}(r, n, C_n, B_n, H)$ from the invariant and open its definition. From the definition of $I_n.out$ and $C_n(k) \neq \bot$ we first obtain $k \notin I_n.out$. Together with the constraint $\phi_1(n, B_n, C_n, I_n)$ we then infer $B_n(k) = C_n(k)$. This leaves us with the proof context on line 17. Now we access the precondition $\mathsf{MCS}(r, t, H)$ of the atomic triple and sync it with the view of H and t in the invariant. Moreover, we use the resource $\boxed{\circ\, C_n}^{\gamma_s}$ to infer $C_n \subseteq H$, which then implies $(k, t') \in H$ (line 18). The call to `unlockNode` returns $\mathsf{N}(r, n, C_n, B_n)$ to the invariant and commits the atomic triple, which concludes this case.

For the second case, we have $C_n(k) = \bot$ and the thread calls `findNext`. Here, we unfold $\mathsf{N}(r, n, C_n, B_n)$ to retrieve $\mathsf{node}(r, n, es, C_n)$ needed to satisfy the precondition of `findNext`. We then end up again with two subcases: one where there is a successor node n' such that $k \in es(n')$,

and the other where no such node exists. Let us consider the first subcase, which is captured by the proof context on line 25. Now, before the thread unlocks n, we have to reestablish the precondition of `traverse` for the recursive call on line 30. To do this, we first open the invariant and retrieve $\mathsf{S}(r, n, C_n, B_n, H)$. From predicate $\mathsf{closed}(es)$, we obtain the resource $\mathsf{inFP}(n')$ as $es(n') \neq \emptyset$. $\mathsf{inFP}(n')$ is the first piece for the precondition of `traverse`. To obtain the remaining pieces, we must retrieve $\mathsf{S}(r, n', C_{n'}, B_{n'}, H)$ from the invariant. This is possible because we can infer $n' \in \mathsf{dom}(I)$ using $\mathsf{inFP}(n')$.

First observe that from $I_n.out(n')(k, B_n(k)) > 0$ and the fact that I_n and $I_{n'}$ compose, we can conclude $I_{n'}.in(n')(k, B_n(k)) > 0$. It then follows from the constraint $\phi_2(n', B_{n'}, I_{n'})$ that $B_{n'}(k) = B_n(k)$. We can further take a fresh snapshot of the resource $\bullet\, B_{n'}(k)^{\gamma_{cir(n')(k)}}$ to obtain our final missing piece $\circ\, B_{n'}(k)^{\gamma_{cir(n')(k)}}$ for the precondition of `traverse`. Then substituting both $B_n(k)$ and $B_{n'}(k)$ by a fresh variable t_1 and folding predicate $\mathsf{N}(r, n, C_n, B_n)$ we arrive at line 27. Unlocking n transfers ownership of $\mathsf{N}(r, n, C_n, B_n)$ back to the invariant. The resulting proof context satisfies the precondition of the recursive call to `transfer`, which we use to commit the atomic triple by applying the specification of `transfer` inductively.

We are left with the last subcase where n has no successor for k (line 32). Here, we proceed similarly to the first case above: we obtain $\mathsf{S}(r, n, C_n, B_n, H)$ from the invariant and use $\forall n'.\ k \notin es(n')$ to conclude that $I_n.out(n')(k, B_n) = 0$ for all n'. It then follows from the constraint $\phi_1(n, B_n, C_n, I_n)$ that $B_n(k) = C_n(k)$. After folding predicate $\mathsf{N}(r, n, C_n, B_n)$ we arrive on line 33. The linearization point in this case is at the point when n is unlocked, so we access the precondition $\mathsf{MCS}(r, t, H)$ of the atomic triple and sync it with the view of H and t in the invariant. Again, using the resource $\circ\, C_n^{\gamma_s}$ we infer $C_n \subseteq H$ to conclude $(k, \bot) \in H$. The call to `unlockNode` returns $\mathsf{N}(r, n, C_n, B_n)$ to the invariant and commits the atomic triple. The postcondition follows for $t' = \bot$, which is the return value in this case.

This completes the proof of search recency for general multicopy structures.

9.2.2 PROVING THE CORRECTNESS OF UPSERT

We now focus on proving the correctness of the `upsert` operation presented in Fig. 9.1. As the code of `upsert` is identical to the version in the two-copy template, the general outline of the Iris proof closely follows the one presented in Section 8.2.2. We therefore only discuss those aspects of the proof that are concerned with our encoding of contents-in-reach using flows.

The `upsert` changes the physical contents C_r of the root node. As a consequence,the contents-in-reach $C_{ir}(r)$ also changes. However, for all other nodes n, $C_{ir}(n)$ stays the same. This is the key observation that enables us to a obtain local proof argument for the correctness of `upsert` in the general case of a multicopy structure with an unbounded number of nodes.

In the proof, the change of $C_{ir}(r)$ must be reflected by an update of the ghost resource that holds B_r in the predicates $\mathsf{N}(r, r, C_r, B_r)$ and $\mathsf{S}(r, r, C_r, B_r, H)$ of the invariant. In turn, this means that the outflow of the interface I_r may need to be updated, as well. We thus have to prove that the

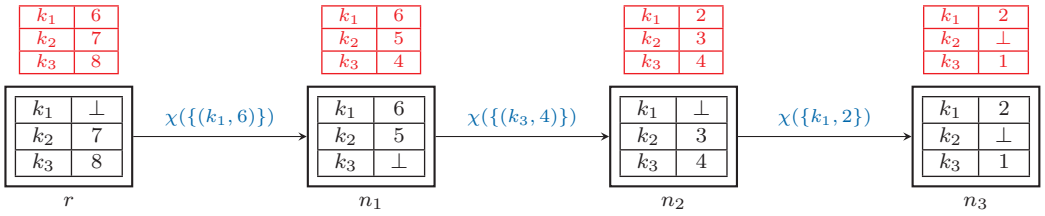

Figure 9.8: Multicopy structure from Fig. 9.4 after upserting $(k_3, 8)$ at the root node.

resulting new interface I'_r of r still composes with the interfaces of the other nodes in the graph and that the relevant constraints imposed on B'_r, C_r, and I'_r by the invariant still hold.

There are several cases we need to cover. To illustrate these, consider again the multicopy structure in Fig. 9.4. The simple case is when the `upsert` modifies an existing copy of the query key in the root. For instance, suppose we upsert key k_2 to timestamp 8 in the structure shown in Fig. 9.4. That is, C_r is updated to $C'_r = C_r[k_2 \rightarrowtail 8]$ and, in turn, B_r is updated to $B'_r = B_r[k_2 \rightarrowtail 8]$. First note that the constraint:

$$I_r.out = \lambda n'.\, \chi(\{(k,t) \mid k \in es(n') \wedge C'_r(k) = \bot \wedge B'_r(k) = t\})$$

still holds for the old interface I_r because we already had $C_r(k_2) \neq \bot$ before the upsert. Hence, we do not have to change I_r in this case, and it still composes with the rest of the graph. Also, it is easy to see that the constraint $\phi_1(r, B'_r, C'_r, I_r)$ remains valid. The constraint $\phi_2(r, B'_r, I_r)$ remains trivially valid because I_r has no inflow. This argument generalizes to all those cases where $C_r(k) \neq \bot$ before the update or where $k \notin es(n')$ for all nodes n'.

The more interesting case is when `upsert` adds a new copy of a key k that was not yet present in the root node and there exists and edge leaving r that has k in its edgeset. For instance, suppose k_3 is upserted to 8 in the multicopy structure shown in Fig. 9.4. In this case, we had $C_r(k_3) = \bot$ before the upsert but we have $C_r(k_3) \neq \bot$ after the upsert. Hence, in order to make sure that the constraint on the outflow of r's interface is still satisfied, we need to change the outflow of I_r. Figure 9.8 shows the updated multicopy structure for the considered `upsert` on k_3 in Fig. 9.4. Note that the outflow of r's interface to n_1 (i.e., the label on the edge between these nodes) is reduced from $\chi(\{(k_1, 6), (k_3, 4)\})$ to $\chi(\{(k_1, 6)\})$.

In general, suppose we have $C_r(k) = \bot$ before the upsert and $k \in es(n')$ for some node n'. In this case, we obtain the new I'_r from I_r by defining

$$\begin{aligned} I'_r.in &:= I_r.in \\ I'_r.out &:= \lambda n'.\ (k \in es(n')\ ?\ I_r.out(n')[(k, B_r(k)) \rightarrowtail 0] : I_r.out(n')) \end{aligned}$$

First note that $\phi_1(r, B'_r, C'_r, I'_r)$ remains valid because $B'_r(k) = C'_r(k)$ for the upserted key k. Moreover, $\phi_2(r, B'_r, I'_r)$ remains valid because the inflow did not change and I_r had no inflow.

Unfortunately, since the outflow of r's interface has changed, the new interface I'_r will no longer compose with the rest of the graph. That is, the interface $I_{n'}$ for the node n' such that $k \in es(n')$ still expects a larger inflow compatible with the old outflow provided by I_r. To account for this, we also need to update the inflow of interface $I_{n'}$. Note that we can do this without locking n' because this change does not violate any of the node-local invariants at n'. This is reflected by the fact that the resource holding $I_{n'}$ remains in the invariant even if n' is locked by another thread. We update $I_{n'}.in$ by accounting for the reduced outflow from r as follows:

$$I'_{n'}.out \coloneqq I_{n'}.out$$
$$I'_{n'}.in \coloneqq \lambda n.\ (n = n'\ ?\ I_{n'}.in(n)[(k, B_r(k)) \rightarrowtail I_{n'}.in(n)(k, B_r(k)) - 1] : I_{n'}.in(n))$$

Note that because the outflow of the interface remains the same, all constraints imposed on $I'_{n'}.out$ by the invariant are still satisfied. Moreover, $\phi_2(n', B_{n'}, I'_{n'})$ continues to hold because the inflow of n' only decreases, i.e., for all keys k' and t, $I'_{n'}.in(n')(k', t) \leq I_{n'}.in(n')(k', t)$.

One can now easily verify that the composite interface of r and n' before and after the upsert is preserved, i.e., $I_r \oplus I_{n'} = I'_r \oplus I'_{n'}$. We can therefore replace I_r and $I_{n'}$ in the invariant by their new interfaces in a single frame-preserving update at the commit point in the proof of `upsert`.

The remainder of the proof closely follows that presented for the two-node template in Section 8.2.2.

9.2.3 PROVING THE CORRECTNESS OF MAINTENANCE

In order to prove correctness of the maintenance operation undertaken by `compact` (Fig 9.2), we must show that the operation satisfies the following specification required by Theorem 7.1 from Chapter 7.

$$\boxed{\mathsf{mcs\_inv}(r)} \mathrel{-\!\!*} \mathsf{inFP}(n) \mathrel{-\!\!*} \langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\rangle\ \texttt{compact n}\ \langle\, \mathsf{MCS}(r, t, H) \,\rangle$$

The above specification says that `compact` logically takes effect in a single atomic step, and at this step the global state of the data structure does not change. Note that for $n = r$, the auxiliary precondition $\mathsf{inFP}(n)$ follows directly from the invariant $\mathsf{mcs\_inv}(r)$.

Technically, the linearization point of the operation occurs when the lock on the final node n is released on line 12, just before the function terminates. However, the interesting part of the proof is to show that the changes to the physical contents of nodes n and m performed by each call to `mergeContents` at line 6 preserve the abstract state of the structure as well as the invariant. In particular, the changes to C_n and C_m also change the contents-in-reach of C_m. We need to argue that this is a local change that does not propagate further in the data structure, similar to our proof of `upsert`.

Updated invariant for proving correctness of maintenance. When proving the correctness of `compact`, we face two technical challenges. The first challenge arises when establishing that `compact` changes the contents of the nodes involved in such a way that the high-level invariants are maintained.

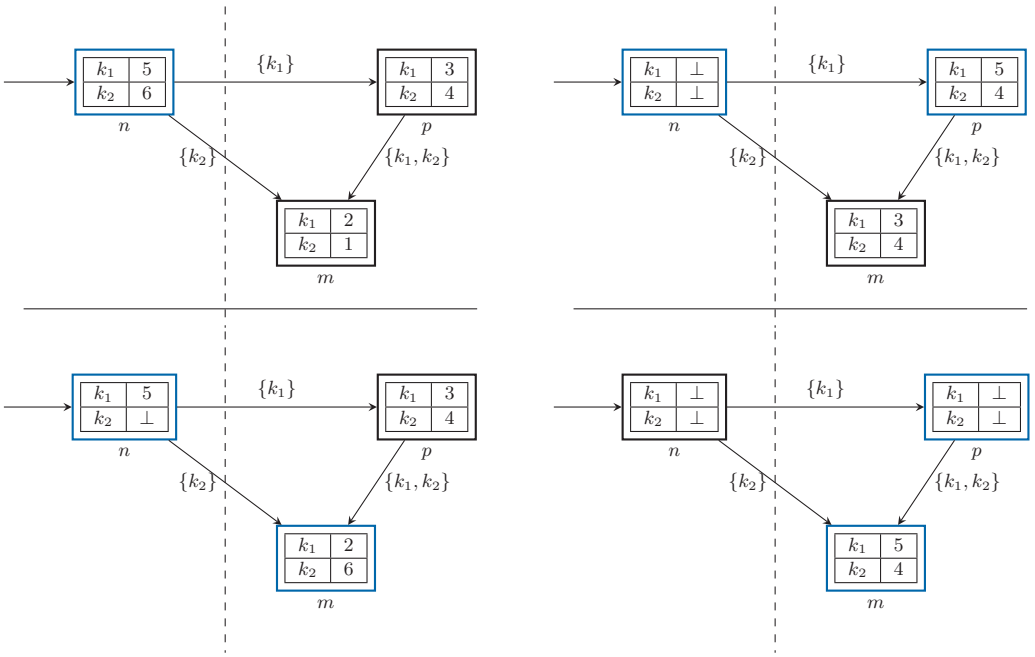

Figure 9.9: Possible execution of the `compact` operation on a DAG. Edges are labeled with their edgesets. The nodes undergoing compaction in each iteration are marked in blue.

In particular, it is difficult to reestablish Invariant 2, which states that the contents-in-reach can only increase over time. Since compaction replaces downstream copies of keys with upstream copies, in order to maintain Invariant 2 we need the additional property that the timestamps of keys in the contents of nodes can only decrease as we move away from the root. This is captured by the following additional invariant:

Invariant 6 At any atomic step, the contents of a node is not smaller than the contents-in-reach of its successor. That is, for all keys k and nodes n and m, if $k \in \mathsf{es}(n, m)$ and $C_n(k) \neq \bot$ then $C_{ir}(m)(k) \leq C_n(k)$.

This invariant is not yet captured by $\mathsf{mcs\_inv}(r)$.

The second challenge is that the maintenance template only generates tree-like structures and this invariant about the data structure graph is critical for the correctness of `compact`. However, the Iris invariant $\mathsf{mcs\_inv}(r)$ presented thus far allows for arbitrary DAGs (in fact, it does not even rule out cycles).

To further motivate these two issues, consider the multicopy structure depicted at the top left of Fig. 9.9. The logical contents of this structure (i.e. the contents-in-reach of n) is $[k_1 \rightarrowtail 5, k_2 \rightarrowtail 6]$.

The structure at the bottom left shows the result obtained after executing `compact` n to completion where n has been considered to be at capacity and the successor m has been chosen for the

merge, resulting in $(k_2, 6)$ being moved from n to m. Note that at this point the logical contents of the data structure is still $[k_1 \rightarrowtail 5, k_2 \rightarrowtail 6]$ as in the original structure. However, the structure now violates Invariant 6 for nodes p and m since $B_m(k_2) > C_p(k_2)$.

Suppose that now a new compaction starts at n that still considers n at capacity. Now node p is chosen for the merge, resulting in $(k_1, 5)$ being moved from n to p. The graph at the top right depicts the resulting structure. The compaction then continues with p, which is also determined to be at capacity. Node m is chosen for the merge, resulting in $(k_1, 5)$ and $(k_2, 4)$ being moved from p to m. At this point, the second compaction terminates. The final graph at the bottom right shows the resulting structure at this point. Observe that the logical contents is now $[k_1 \rightarrowtail 5, k_2 \rightarrowtail 4]$, violating the specification which implies that the logical contents is to be preserved by maintenance. In fact the contents-in-reach of n has decreased, violating Invariant 2.

The example illustrates the importance of the tree invariant for the correctness of the maintenance template. This invariant is needed to guarantee that Invariant 6 is maintained, which in turn is needed to maintain Invariant 2 when copies of keys are merged downstream. We observe that, although `compact` will only create tree-like structures, we can prove its correctness using a weaker invariant about the structure of the generated graphs, namely that at all times, every node has at most one incoming edge. We will use this weaker invariant in our proof below.

To capture the two additional invariants, we define two new flows which we track using additional flow interfaces in $\mathsf{mcs\_inv}(r)$ similar to the flow interface I used to encode contents-in-reach.

First we start with the flow used to express Invariant 6. In the following, we will use the variable J to refer to interfaces related to this flow. At the high-level, the idea to capture Invariant 6 is for a node to propagate its contents to its successors. As a result, the inflow of the node-level interface J_m of a node m encodes the union of the contents of all its predecessor nodes n. We can then express Invariant 6 as a node-local condition that relates $J_m.in$ and B_m.

More concretely, the new global interface J is stored at a new ghost location γ_J in $\mathsf{mcs\_inv}(r)$. As for the contents-in-reach flow interface I, the associated RA is authoritative flow interfaces over the flow domain of multisets of key-timestamp pairs. We add the constraint $\mathsf{dom}(I) = \mathsf{dom}(J)$ to ensure that I and J agree on which nodes belong to the graph. We further demand that J has no inflow.

We next augment $\mathsf{S}(r, n, C_n, B_n)$ with the fragmental ownership $\circ J_n^{\gamma_J}$ of the corresponding singleton interface J_n. The desired flow is then obtained by adding a constraint defining the outflow of J_n as

$$J_n.out = \lambda n'.\, \chi(\{(k,t) \mid k \in es_n(n') \wedge C_n(k) = t \wedge t \neq \bot\})$$

Finally, to capture Invariant 6, we add the following predicate as an additional conjunct to the definition of $\mathsf{S}(r, n, C_n, B_n)$:

$$\phi_3(n, B_n, J_n) \coloneqq \forall k\, t.\, I_n.in(n)(k,t) > 0 \Rightarrow t \leqslant B_n(k) \tag{9.4}$$

Next, let us tackle the invariant that each node has at most one predecessor. The high-level idea to encode this invariant is to perform reference counting using flows. For this purpose, we

introduce one more global interface to $\mathsf{mcs\_inv}(r)$ which we refer to by the variable R and store at ghost location γ_R. The underlying flow domain here is natural numbers with addition as the monoid operation. Again we add the conjunct $\mathsf{dom}(R) = \mathsf{dom}(I)$ to $\mathsf{mcs\_inv}(r)$ in order to ensure that the interfaces agree on which nodes belong to the graph.

Now, by demanding that R has no inflow and by letting every node propagate outflow of 1 to each of their successors, the flow of every node will be equal to the number of its incoming edges. For the encoding, we again augment $\mathsf{S}(r, n, C_n, B_n)$ with the fragmental ownership $\overset{\gamma_R}{\circ}\, R_n$ of the corresponding singleton interface R_n and add a constrain defining the desired outflow:

$$R.out \coloneqq \lambda n'.\ (es_n(n') \neq \emptyset\ ?\ 1 : 0)$$

The structural invariant is then captured by conjoining $\mathsf{S}(r, n, C_n, B_n)$ with the predicate:

$$\phi_4(n, R_n) \coloneqq R_n.in(n) \leqslant 1 \tag{9.5}$$

We briefly explain why we can still prove the correctness of `search` and `upsert` with the updated invariant $\mathsf{mcs\_inv}(r)$. First note that `search` does not modify any ghost resources. So the new invariant is still trivially maintained. Moreover, we are still able to follow the same proof outline from Fig. 9.7 by simply ignoring the newly added ghost resources related to the interfaces J and R.

Now let us consider the operation `upsert` $r\ k$. Since `upsert` does not change the edgesets of any nodes, the resources and constraints related to the reference counting flow interfaces R_n are trivially maintained. However, since the contents of the root node r is changed, the interfaces J_r as well as the interface J_n for any successor n of r with $k \in es_r(n)$ must be updated to include the new pair $(k, C'_r(k))$. The contents of r here changes to $C'_r = C_r[k \rightarrowtail t]$ where t is the current clock value. The predicate $\mathsf{maxTS}(H)$ in the invariant guarantees that $C'_r(k) = t$ is larger than all previous timestamps in the structure. Hence, propagating (k, t) to a successor n of r will preserve the constraint $\phi_3(n, B_n, J_n)$.

With the updated definition of $\mathsf{mcs\_inv}(r)$, we are now ready to prove the correctness of `compact`.

High-level proof of `compact`. The specifications of the implementation-specific helper functions assumed by `compact` are provided in Fig. 9.10. A thread performing `compact n` starts by locking node n and checking if node n is at full capacity using the helper function `atCapacity`. By locking node n, the thread receives the resources available in $\mathsf{N}(r, n, C_n, B_n)$, for some contents C_n and contents-in-reach B_n. The precondition of `atCapacity` requires the predicate $\mathsf{node}(r, n, es_n, C_n)$, which is available to the thread as part of $\mathsf{N}(r, n, C_n, B_n)$. The return value of `atCapacity n` is a boolean indicating whether node n is full or not. The precise logic of how the implementation of `atCapacity` determines whether a node is full is immaterial for the correctness of the template, so the specification of `atCapacity` abstracts from this logic. If n is not full, `compact` releases the lock on n, transferring ownership of $\mathsf{N}(r, n, C_n, B_n)$ back to the invariant and then terminates. The call to `unlockNode` on line 12 is the commit point of the atomic triple in the else branch of the conditional.

1 $\{\mathsf{node}(r, n, es, C_n)\}$
2 `atCapacity n`
3 $\{b.\ \mathsf{node}(r, n, es, C_n) * b = true \vee b = false\}$
4
5 $\{\mathsf{node}(r, n, es_n, C_n)\}$
6 `chooseNext n`
7 $\left\{\begin{array}{r} m.\ \mathsf{node}(r, n, es'_n, C_n) * es'_n(m) \neq \emptyset \\ * (es_n = es'_n \vee es'_n = \lambda_\emptyset[m \rightarrowtail \mathsf{KS}] * \mathsf{node}(r, m, es_m, C_m) * C'_n = \lambda_\perp * es_m = \lambda_\emptyset) \end{array}\right\}$
8
9 $\{\mathsf{node}(r, n, es_n, C_n) * \mathsf{node}(r, m, es_m, C_m) * es_n(m) \neq \emptyset\}$
10 `mergeContents n m`
11 $\left\{\begin{array}{r} \mathsf{node}(r, n, es_n, C'_n) * \mathsf{node}(r, m, es_m, C'_m) * C'_n \subseteq C_n * C'_m \subseteq C_n \cup C_m \\ * merge(C_n, C_m) = merge(C'_n, C'_m) \end{array}\right\}$

Figure 9.10: Specifications of helper functions used by `compact` - `atCapacity n`, `chooseNext n`, and `mergeContents n m`.

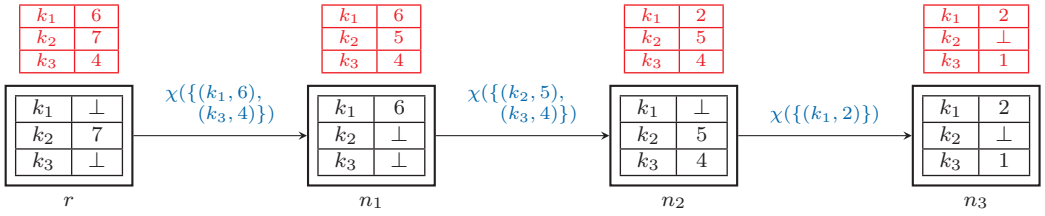

Figure 9.11: Total merge of n_1 with n_2 in multicopy structure shown in Fig. 9.4.

Thus, let us consider the other case, i.e. when n is full. Here, the contents of node n must be merged with the contents of some successor node m of n. This node m is determined by the helper function `chooseNext`. `chooseNext` either returns an existing successor m of n (i.e., $es'_n(m) \neq \emptyset$ and $es_n = es'_n$), or the returned node m is a freshly allocated node linked into graph such that it is now a successor of n. In the former case, we can establish that m is part of the data structure due to the fact that the edgeset of n directs some keys to m. This follows from the property $\mathtt{closed}(n)$ in the invariant $\mathsf{mcs\_inv}(r)$. In the latter case, we must extend the domain of the global flow interface tracked by the invariant with the new node m. This can be done using a frame-preserving update of the authoritative version of the interfaces at ghost locations γ_I, γ_J, and γ_R (see rule FLOWINT-DOM-UPD in §6.1.2). Showing that the invariant is preserved is easy because the invariant guarantees that m can only be reached via m. Overall, once it has been established that node m is in the footprint of the data structure, m is locked and the contents of n is (partially) merged into m using the helper function `mergeContents` (line 6).

Let us now examine the specification of the function `mergeContent` in detail. `mergeContents` n m merges the data from node n into node m. By merge, we mean that some copies

of keys are transferred from n to m, possibly replacing older copies in m. Figure 9.11 shows the merge process on the multicopy structure from Fig. 9.4. Here, the copy $(k_2, 5)$ has been transferred from node n_1 to n_2. In general, `mergeContents` modifies the contents of the two nodes according to the specification given in Fig. 9.10. The precondition demands ownership of the physical representation of the nodes' contents and that m is a successor of n. The contents are modified to C'_n and C'_m respectively such that $C'_n \subseteq C_n$ and $C'_m \subseteq C_n \cup C_m$ (i.e., no new copies are conjured out of thin air). To further constrain the new contents of the nodes, the postcondition additionally demands $merge(C_n, es_n(m), C_m) = merge(C'_n, es_n(m), C'_m)$, where

$$merge(C_n, Es, C_m) \coloneqq \lambda k. \begin{cases} C_n(k) & \text{if } C_n(k) \neq \bot \\ C_m(k) & \text{else if } k \in Es \\ \bot & \text{otherwise} \end{cases}$$

We next show that, together, these constraints ensure that we can consistently update all relevant ghost resources in the invariant. In particular, we will show that the contents-in-reach of m can only increase and that the contents-in-reach of all other nodes, including n remain unchanged.

For any key k and contents C, we denote by $\mathsf{dom}(C)$ the set of keys k such that $C(k) \neq \bot$. The set $K \coloneqq \mathsf{dom}(C_n) \setminus \mathsf{dom}(C'_n)$ then denotes all keys whose copies are merged from C_n into C_m.

Observe that the last conjunct in the postcondition of `mergeContents` guarantees that only copies of keys in the edgeset of the edge (n, m) are merged. That is, if $k \in K$, then $k \in es_n(m)$. This is true because if $k \in \mathsf{dom}(C_n) \setminus \mathsf{dom}(C'_n)$ and $k \notin es_n(m)$, then $merge(n, C'_n, es_n(m), C'_m)(k) = \bot$, but $merge(n, C_n, es_n(m), C_m)(k) = C_n(k) \neq \bot$, which is a contradiction. We will use this observation freely in the remainder of the proof.

Before we can proceed with the rest of the proof, we need to choose witnesses for the existentially quantified variables in the invariant $\mathsf{mcs\_inv}(r)$ before the call to `mergeContents`. For a node p in the structure, we denote by I_p, J_p, and R_p the fragmental singleton flow interface of node p at ghost locations γ_I, γ_J, and γ_R, respectively. Moreover, let B_p be the set stored at ghost location $\gamma_{b(p)}$ (i.e., the contents-in-reach of p before the call to `mergeContents`).

First, since the update of C_m also affects the contents-in-reach of m, we need to update B_m appropriately. We do this by defining:

$$B'_m \coloneqq \lambda k.\ (k \in K\ ?\ C_n(k) : B_m)$$

and then replace B_m at ghost location $\gamma_{b(m)}$ by B'_m. Additionally, we need to update each ghost location $\gamma_{cir(m)(k)}$ to hold the new value $B'_m(k)$. To do this, the authoritative maxnat RA requires us to show that $B_m(k) \leq B'_m(k)$. If $k \notin K$, then $B_m(k) = B'_m(k)$ by definition. Hence consider the case where $k \in K$. We then have $B'_m(k) = C_n(k) \neq \bot$. From this and $k \in e_n(m)$, it follows that $J_n.out(m)(k, C_n(k)) > 0$, which using the flow equation enables us to conclude $J_n.in(m)(k, C_n(k)) > 0$. We then infer from $\phi_3(m, B_m, J_m)$ that $B_m(k) \leq C_n(k) = B'_m(k)$.

Next, the fragmental singleton interfaces I_p for the successors p of m are affected by the update of m. Hence, they need to be updated together with the interfaces of n and m. To this end, let $S_m \coloneqq \{p \mid es_m(p) \neq \emptyset\}$ be the set of all successors of m. Since C_n and C_m change, we need to update the outflows of I_n and I_m, respectively, the inflows of I_m and I_p for all $p \in S_m$ accordingly. We, thus, define the new node-level interfaces of these nodes to be consistent with the new contents C'_n and C'_m:

$$
\begin{aligned}
I'_n.in &\coloneqq I_n.in \\
I'_n.out &\coloneqq \lambda n'\,(k,t).\, I_n.out(n')(k,t) + (n' = m \wedge k \in K \wedge C_n(k) = t \;?\; 1 : 0) \\
I'_m.in &\coloneqq \lambda n'\,(k,t).\, I_m.in(n')(k,t) + (n' = m \wedge k \in K \wedge C_n(k) = t \;?\; 1 : 0) \\
I'_m.out &\coloneqq \lambda n'\,(k,t).\, I_m.out(n')(k,t) - (k \in es_m(n') \wedge k \in K \setminus \mathsf{dom}(C_m) \wedge B_m(k) = t \;?\; 1 : 0) \\
I'_p.in &\coloneqq \lambda n'\,(k,t).\, I_p.in(n')(k,t) - (k \in es_m(n') \wedge k \in K \setminus \mathsf{dom}(C_m) \wedge B_m(k) = t \;?\; 1 : 0) \\
I'_p.out &\coloneqq I_p.out
\end{aligned}
$$

First, note that the changes to the inflows and outflows match up consistently. One can therefore easily verify that the old and new singleton interfaces compose to the same larger interface:

$$I_n \oplus I_m \oplus \bigoplus_{p \in S_m} I_p = I'_n \oplus I'_m \oplus \bigoplus_{p \in S_m} I'_p$$

This means that we can simultaneously replace all old interfaces by their new ones using a frame-preserving update of ghost location γ_I.

We next prove that the new interfaces continue to satisfy the constraints imposed by the invariant. Let us first consider the node n. We start by showing:

$$I'_n.out = \lambda n'.\, \chi(\{(k,t) \mid k \in es(n') \wedge C'_n(k) = \bot \wedge B_n(k) = t\})$$

To see this, observe that $C'_n(k) \neq C_n(k)$ iff $k \in K$, in which case $C'_n(k) = \bot$ and $B_n(k) = C_n(k)$.

Next, note that $\phi_2(n, B_n, I'_n)$ continues to hold because the inflow of n's interface does not change. To prove that $\phi_1(n, B_n, C'_n, I'_n)$ still holds, first note that if $k \notin K$ then $C_n(k) = C'_n(k)$. Moreover, $\phi_1(n, B_n, C_n, I_n)$ implies $C_n(k) = B_n(k) \vee k \in \mathsf{outs}(I_n)$. Since the outflow of I'_n increases from I_n, we also have $C'_n(k) = B_n(k) \vee k \in \mathsf{outs}(I'_n)$, which proves this case. On the other hand, if $k \in K$, then we immediately have by definition of $I'_n.out$ that $k \in \mathsf{outs}(I'_n)$.

We next prove that all constraints are preserved for $p \in S_m$. From the fact that the outflow of p's interface does not change, we can immediately conclude that $\phi_1(p, B_p, C_p, I'_p)$ continues to hold. Moreover, because the inflow of I'_p decreases from I_p, $\phi_2(p, B_p, I'_p)$ also remains true.

Let us thus consider the node m. We first prove that $\phi_1(m, B'_m, C'_m, I'_m)$ holds. We only need to consider two cases since the outflow of I'_m reduces from I_m. The first case is when $k \in \mathsf{outs}(I_m)$ but $k \notin \mathsf{outs}(I'_m)$. In this case, we must have $k \in K \setminus \mathsf{dom}(C_m)$. That is, we have $C'_m(k) = C_n(k) = B'_m(k)$. The only other case we need to consider is when $k \notin \mathsf{outs}(I_m)$ and $k \notin \mathsf{outs}(I'_m)$. In this case, we must have $k \notin K \setminus \mathsf{dom}(C_m)$. If $k \notin K$, then we have $C'_m(k) = C_m(k) = B_m(k) = B'_m(k)$ by $\phi_1(m, B_m, C_m, I_m)$. On the other hand, if $k \in K$, then we immediately have $C'_m(k) = C_n(k) = B'_m(k)$.

To prove that $\phi_2(m, B'_m, I'_m)$ holds, suppose that $I'_m.in(m)(k,t) > 0$. First, consider the case where $I_m.in(m)(k,t) = 0$. Together with $I'_m.in(m)(k,t) > 0$, this implies $k \in K$ and $C_n(k) = t$. We therefore immediately have $B'_m(k) = C_n(k) = t$. Thus, consider the case where $I_m.in(m)(k,t) > 0$ but assume $B'_m(k) \neq t$ for the sake of deriving a contradiction. First, from $\phi_2(m, B_m, I_m)$ it then follows that $B_m(k) = t$. From this we conclude $B'_m(k) \neq B_m(k)$ which implies $k \in K$ and, hence, $C_n(k) \neq \bot$. It follows that $I_n.out(m)(k,t) = 0$. Because the global interface I has no inflow, we can infer that there must exist some other node n' such that $I_{n'}.out(m)(k,t) > 0$. This, in turn, implies $k \in es_{n'}(m)$. Hence, we must also have $R_{n'}.out(m) = 1$. Since $k \in K$ we further have $k \in es_n(m)$, which also implies $R_n.out(m) = 1$. It then follows from the flow equation that $R_m.in(m) \geq R_n.out(m) + R_{n'}.out(m) = 2$ which contradicts $\phi_4(m, R_m)$.

We similarly need to construct new fragmental singleton interfaces at ghost location γ_J that reflect the updates to C_n and C_m. Specifically, the outflows of J_n and J_m, respectively, the inflows of J_m and J_p for all $p \in S_m$ are affected by these updates. We define the new interfaces for these nodes appropriately as follows:

$$
\begin{aligned}
J'_n.in &\coloneqq J_n.in \\
J'_n.out &\coloneqq \lambda n'\,(k,t).\, J_n.out(n')(k,t) - (n' = m \wedge k \in K \wedge C_n(k) = t \;?\; 1 : 0) \\
J'_m.in &\coloneqq \lambda n'\,(k,t).\, J_n.in(n')(k,t) - (n' = m \wedge k \in K \wedge C_n(k) = t \;?\; 1 : 0) \\
J'_m.out &\coloneqq \lambda n'\,(k,t).\, J_m.out(m')(k,t) \; - (k \in es_m(n') \wedge k \in K \wedge C_m(k) = t \;?\; 1 : 0) \\
&\qquad\qquad\qquad\qquad + (k \in es_m(n') \wedge k \in K \wedge C_n(k) = t \;?\; 1 : 0) \\
J'_p.in &\coloneqq \lambda n'\,(k,t).\, J_p.in(n')(k,t) \; - (n' = p \wedge k \in es_m(n') \wedge k \in K \wedge C_m(k) = t \;?\; 1 : 0) \\
&\qquad\qquad\qquad\qquad + (n' = p \wedge k \in es_m(n') \wedge k \in K \wedge C_n(k) = t \;?\; 1 : 0) \\
J'_p.out &\coloneqq J_p.out
\end{aligned}
$$

Again, it is easy to verify that these new interfaces compose to the same large interface as the old interfaces and can, hence, replace the old ones in a frame-preserving update. Proving that the new interfaces continue to satisfy the constraints imposed by the invariant $\mathsf{mcs\_inv}(r)$ is also straightforward. We only consider the most interesting case here: proving that $\phi_3(p, B_p, J'_p)$ continues to hold. For this, it suffices to show that if $k \in K$ and $k \in es_m(p)$, then $B_p(k) \leq C_n(k)$. To see this, first note that $k \in K$ implies $C_n(k) \neq \bot$ and $k \in es_n(m)$. It follows that $J_n.out(m)(k, C_n(k)) > 0$ and hence $J_m.in(m)(k, C_n(k)) > 0$. Thus, we know from $\phi_3(m, B_m, J_m)$ that $B_m(k) \leq C_n(k)$. Now we consider two subcases. If $C_m(k) = \bot$, then $I_m.out(p)(k, B_m(k)) > 0$ and hence $I_p.in(p)(k, B_m(k)) > 0$. Then it follows from $\phi_2(p, B_p, I_p)$ that $B_p(k) = B_m(k)$ and we can conclude $B_p(k) \leq C_n(k)$. On the other hand, if $C_m(k) \neq \bot$, then $B_m(k) = C_m(k)$. Moreover, we obtain $I_m.out(p)(k, C_m(k)) > 0$ which implies $I_m.in(p)(k, C_m(k)) > 0$. We then obtain from $\phi_3(p, B_p, J_p)$ that $B_p(k) \leq C_m(k) = B_m(k)$. By transitivity, we finally conclude $B_p(k) \leq B_m(k) \leq C_n(k)$.

CHAPTER 10

Related Work, Future Work, and Conclusion

10.1 RELATED WORK

Our work builds on the search structure templates of Shasha and Goodman [1988], the Iris separation logic [Jung et al., 2018], and the flow framework [Krishna et al., 2018, 2020b]. Our main technical contributions relative to these works are a new proof technique for verifying template algorithms of concurrent search structures that relies on the integration of the flow framework into Iris. The notion of edgesets and keysets are taken from Shasha and Goodman [1988] but we show how to reason locally about them using flows. Specifically, we capture the essence of the Keyset Theorem of Shasha and Goodman [1988] in terms of an Iris RA, thereby eliminating any dependencies on a specific programming language semantics, and allowing us to easily mechanize the proof in Iris. We also provide a full mechanization of the meta-theory of the flow framework presented in [Krishna et al., 2020b] in Coq/Iris and GRASShopper. We note that Krishna et al. [2018] use the flow framework to verify a template algorithm based on the give-up technique. However, their proof is only on paper, still depends on a meta-level Keyset Theorem like [Shasha and Goodman, 1988] and uses a bespoke program logic that is difficult to mechanize due to limitations of the original flow framework (cf. [Krishna et al., 2020b]).

To our knowledge, we are the first to provide a mechanized proof of a concurrent B-link tree. Unlike the proof of da Rocha Pinto et al. [2011], which is not mechanized, our proof does not assume node-level operations to be given as primitives. In particular, we also verify the challenging split operation. The only other comparable proof is that of a B+ tree in [Malecha et al., 2010]. However, this work only considers a sequential B-tree implementation and the proof is considerably more complex than ours (encompassing more than 5000 lines of proof for roughly 500 lines of code). Moreover, much of our proof can be reused to verify other concurrent search structures that rely on linking, such as the concurrent hash table implementation that we consider.

Feldman et al. [2018] show how to simplify linearizability proofs of concurrent data structures with unsynchronized searches by reasoning purely sequentially about the traversal performed by the search. Their contribution is orthogonal to ours as they do not aim to parameterize the concurrency proof by the heap representation of the data structure.

Iris does not support reasoning about deallocation. Therefore our proofs assume a garbage collected environment. However, Meyer and Wolff [2019] demonstrate a similar proof modularity by decoupling the proof of data structure correctness from that of the underlying memory reclama-

tion algorithm, allowing the correctness proof to be carried out under the assumption of garbage collection. An alternative approach to extending our proofs to deal with memory reclamation is to use Iron [Bizjak et al., 2019], a recent extension of Iris that allows proving absence of memory leaks. It is a promising direction of future work to integrate these approaches and our technique in order to obtain verified data structures where the user can mix-and-match the synchronization technique, memory layout, and the memory reclamation algorithm.

There exist many other program logics that help modularize the correctness proofs of concurrent systems [Bornat et al., 2005, da Rocha Pinto et al., 2014, Dinsdale-Young et al., 2010, Feng et al., 2007, Gu et al., 2018, Heule et al., 2013, Nanevski et al., 2014, Raad et al., 2015, Vafeiadis and Parkinson, 2007, Xiong et al., 2017]. Like Iris, their main focus is on modularizing proofs along the interfaces of components of a system (e.g. between the client and implementation of a data structure) and accounting for differences in the concurrency semantics across different abstraction layers [Gu et al., 2018]. Instead, we focus on modularizing the proof of a single component (a concurrent search structure) so that the parts of the proof can be reused across many diverse implementations.

As discussed in §6.5, lock-free implementations of search structures often have non-fixed as well as external linearization points. Much work has been dedicated to addressing this challenge [Bouajjani et al., 2013, 2017, Chakraborty et al., 2015, Delbianco et al., 2017, Dodds et al., 2015, Frumin et al., 2018, Khyzha et al., 2017, Liang and Feng, 2013, O'Hearn et al., 2010, Zhu et al., 2015]. However, we note that these papers do not aim to separate the proof of thread safety from the proof of structural integrity. In fact, we see our contributions as orthogonal to these works. For example, we can build on the recent work of supporting prophecy variables in Iris [Jung et al., 2020] to extend our methodology to non-blocking algorithms, as we discuss below.

We have carried out our proof mechanization effort in Iris [Jung et al., 2016, 2018, 2015, Krebbers et al., 2017] and GRASShopper [Piskac et al., 2014]. Iris' support for user-definable ghost resources has been particularly helpful in the mechanization of our template proofs and GRASShopper's proof automation allowed us to scale our verification of implementations to real-world concurrent search structures such as the B-link tree. However, the presented approach is not bound to these specific tools. First, our proof methodology can be replicated in other separation logics that support user-defined ghost state, such as FCSL [Sergey et al., 2015], which would also be useful if one wanted to extend this work to non-linearizable data structures [Sergey et al., 2016]. Second, we already have preliminary experience in automating flow-based proofs in the SL-based verification system Viper [Müller et al., 2017]. Moreover, many of the ideas and techniques proposed in this book can be used in other verification systems, including systems that are not based on separation logic such as Dafny [Leino, 2010, 2017] and CIVL [Hawblitzel et al., 2015].

Fully automated proofs of linearizability by static analysis and model checking have been mostly confined to simple list-based data structures [Abdulla et al., 2013, Amit et al., 2007, Bouajjani et al., 2015, Cerný et al., 2010, Dragoi et al., 2013, Vafeiadis, 2009]. Recent work by Abdulla et al. [2018] shows how to automatically verify more complex structures such as concurrent skip lists that combine lists and arrays. Our work shows that it is possible to devise semi-automated techniques (in

which one formulates useful invariants) that work over a broad class of diverse data structures. Full automation for such structures is still beyond the state of the art.

10.2 FUTURE WORK

Combining the edgeset framework with the flow framework has allowed us to mechanically prove the safety of single-copy and multicopy search structures. We have shown such proofs for B-link trees, hash-tables, and log-structured merge (LSM) trees, among others. Here are some directions in which our work can be extended:

1. Generalizing the presented templates to cover other existing or potential search structure algorithms.
2. Extending the proof technique to prove liveness properties such as termination or deadlock-freedom.
3. Verifying templates that use lock-free concurrent techniques.

We will now examine these directions in turn.

10.2.1 GENERALIZATIONS AND EXTENSIONS

Our discussions of multicopy structures began with a graph model, but we have limited our discussion to list-based LSM trees. Our template algorithms, however, would carry over essentially unchanged to a directed acyclic graph structure with the root in memory as now, but branching on disk. A further generalization would be to allow concurrent updates to the disk nodes when doing compaction, using single node concurrency techniques. Our framework offers the tools for these generalizations.

10.2.2 PROVING LIVENESS

Proving liveness properties such as progress, deadlock-freedom, or termination would involve strengthening the invariants we have used for our template algorithms. For example, the invariant that the keysets of any two nodes in a single-copy structure are disjoint was sufficient to prove partial correctness, but a liveness property like termination would require the stronger invariant that the set of keysets of all nodes cover the key space. This captures the high-level property that given any key k, there is some node in the structure that is responsible for k.

Proving that such an invariant is maintained would require us to prove that the underlying structure is, in some sense, acyclic. More precisely, we need to show that given any key k, its search path (i.e. the path obtained by starting at the root and following edges that have k in their edgeset) is acyclic. Note that none of the single-copy structure proofs in this book proved such an acyclicity property; for partial correctness, it is sufficient to show that if an operation on k operates on a node n then k is in n's keyset. Such an acyclicity invariant can be encoded by using the effectively-acyclic extension of the flow framework [Krishna et al., 2020b].

Liveness proofs usually proceed by defining a *ranking function*, a function from states of the structure to some well-ordered set. Once we have shown that the search structure is acyclic, we can define the ranking function of an operation as the cardinality of the search path. Acyclicity implies that the search path is a list, which means that every time the operation moves from one node to the next, the number of nodes in the search path reduces.

Note that when reasoning about liveness, it is very important to specify the environment under which the algorithm is operating. For instance, even in an acyclic structure, a search operation on k might never terminate if it is operating in an environment where maintenance operations are continually pre-empting the search operation and performing splits that increase k's search path. Standard assumptions about the environment or scheduler, such as fair scheduling or bounded interference, would be required to rule out such undesirable executions.

Formalizing liveness properties in separation logic would require us to use logics that formalize notions of fairness and ranking functions, such as TaDA-live [D'Osualdo et al., 2019]. Our proof technique can be transferred to such logics without much technical difficulty because our flow-based encoding can be performed in any separation logic that supports user-defined resource algebras. There are also automated tools for proving that algorithms are non-blocking [Gotsman et al., 2009], and it would be interesting to see if we can incorporate our approach with these tools.

10.2.3 LOCK-FREE CONCURRENT SEARCH STRUCTURE ALGORITHMS

In order to be able to use sequential reasoning in node-level operations, we have assumed the availability of a locking mechanism. However, many lock-free algorithms have been proposed, e.g. [Harris, 2001, Levandoski et al., 2013] and analyzed as noted in the related work section [Bouajjani et al., 2013, 2017, Chakraborty et al., 2015, Delbianco et al., 2017, Dodds et al., 2015, Frumin et al., 2018, Khyzha et al., 2017, Liang and Feng, 2013, O'Hearn et al., 2010, Zhu et al., 2015].

Very often, the template algorithms we propose continue to apply. For example, a lock-free list in which a thread prepends upserts to the beginning of the list by doing a CAS (compare-and-set) to the root pointer of the list is essentially a multi-copy structure (e.g. in a BW-tree [Levandoski et al., 2013]. Thus it enjoys the same basic invariants as the LSM tree: the logical value of a key is the value associated with the element closest to the head of the list. Properties like search recency still hold.

In fact, an algorithm designer can often use compare-and-swap or the more general compare-and-set in a similar way to locks. Suppose, for example, that a thread records the state of some pointer at time t_1 and then does a compare-and-set based on that recorded state at time t_2. If the compare-and-set succeeds, then the pointer has not changed from t_1 to t_2. Note that if the thread had instead locked the pointer from t_1 to t_2, then the thread would enjoy the same guarantee.

In the prepending example above, "locking the root pointer" plays essentially the same role as "recording the address in the root pointer, constructing a node to prepend, and then doing a compare-and-swap on the root pointer". The advantage of locking is that there is no need to redo work, as there would be if the compare-and-swap fails. The advantage of the compare-and-swap is that lock-

freedom avoids the possibility that a hung thread stops other threads from making progress, because that hung thread holds a lock.

The bottom line is that the *good state* conditions (i.e. the keysets of different nodes are disjoint, and the contents of each node is a subset of its keyset) still apply.

Consider, for example, the following simple (though very inefficient) single-copy lock-free concurrent tree algorithm.

- The root pointer is stored at a fixed location r. Each value of the address stored in that pointer is associated with the time of the last successful modification of the tree.
- A search at time t starts at r and follows the pointer to the copy of that tree structure present as of time t.
- Any modification (i) reads the address A stored in r which is the address of the root of the tree at that time; (ii) copies the existing tree structure and creates a whole new tree structure whose tree root node is at address A'[1]; (iii) performs its modification on the new tree; (iv) uses a compare-and-swap to check whether r still contains the address A and, if so, swaps it to address A'. If r no longer contains address A, the modification must start over.
- The linearization point of each modification is the time that r was successfully modified. The linearization point of every search is the time the search begins.

Of course, copying the entire search structure for every modification is extremely inefficient. An alternative is to (i) copy just the nodes in the path from the root to the leaf that is modified (say in a B-tree like structure), (ii) modify whichever nodes in that path that need to be modified, and then (iii) use a compare and swap to point to the new tree root. No nodes outside that path need be copied.

These methods work well with our proof methodology. The good state conditions are preserved by these algorithms and a thread that begins a search for k at time t and later visits a node n will enjoy the invariant that k is in the inset of n for the structure as of time t. Thus the search's linearization point will be the time when the search begins, i.e., t.

Other lock-free algorithms such as the Harris list [Harris, 2001] (which maintains a list sorted by key) are more fine-grained and require new invariants, which we will address in future work.

10.3 CONCLUSION

We have presented a proof technique for concurrent search structures that (i) separates the reasoning about thread safety from memory safety using the edgeset framework; and (ii) allows local reasoning about global predicates using the flow framework.

We have demonstrated our technique by formalizing and verifying template algorithms using this technique, have shown how to derive verified implementations on specific data structures,

[1] To ensure that tree root node addresses are never repeated, we might require that $A' > A$.

and have thereby mechanized the proofs of linearizability and memory safety for a large class of concurrent search structures.

We believe that this same decompositional approach can be used for other concurrent graph problems in communication networks, memory management, and other domains that use fine-grained concurrent data structures.

Bibliography

Abdulla, P. A., Haziza, F., Holík, L., Jonsson, B., and Rezine, A. (2013). An integrated specification and verification technique for highly concurrent data structures. In Piterman, N. and Smolka, S. A., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 324–338. Springer.

Abdulla, P. A., Jonsson, B., and Trinh, C. Q. (2018). Fragment abstraction for concurrent shape analysis. In Ahmed, A., editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 442–471. Springer.

Amit, D., Rinetzky, N., Reps, T. W., Sagiv, M., and Yahav, E. (2007). Comparison under abstraction for verifying linearizability. In Damm, W. and Hermanns, H., editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer.

Apache Software Foundation (2020a). Apache cassandra. https://cassandra.apache.org/. Last accessed on August 15, 2020.

Apache Software Foundation (2020b). Apache HBase. https://hbase.apache.org/. Last accessed on August 17, 2020.

Bansal, K., Reynolds, A., King, T., Barrett, C. W., and Wies, T. (2015). Deciding local theory extensions via e-matching. In Kroening, D. and Pasareanu, C. S., editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 87–105. Springer.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Bizjak, A., Gratzer, D., Krebbers, R., and Birkedal, L. (2019). Iron: managing obligations in higher-order concurrent separation logic. *Proc. ACM Program. Lang.*, 3(POPL):65:1–65:30.

Bornat, R., Calcagno, C., and Yang, H. (2005). Variables as resource in separation logic. In Escardó, M. H., Jung, A., and Mislove, M. W., editors, *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2005, Birmingham, UK, May*

18-21, 2005, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 247–276. Elsevier.

Bouajjani, A., Emmi, M., Enea, C., and Hamza, J. (2013). Verifying concurrent programs against sequential specifications. In Felleisen, M. and Gardner, P., editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 290–309. Springer.

Bouajjani, A., Emmi, M., Enea, C., and Hamza, J. (2015). On reducing linearizability to state reachability. In Halldórsson, M. M., Iwama, K., Kobayashi, N., and Speckmann, B., editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 95–107. Springer.

Bouajjani, A., Emmi, M., Enea, C., and Mutluergil, S. O. (2017). Proving linearizability using forward simulations. In Majumdar, R. and Kuncak, V., editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer.

Brookes, S. (2007). A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270.

Brookes, S. and O'Hearn, P. W. (2016). Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65.

Burckhardt, S., Alur, R., and Martin, M. M. K. (2007). Checkfence: checking consistency of concurrent data types on relaxed memory models. In Ferrante, J. and McKinley, K. S., editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 12–21. ACM.

Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., and Alur, R. (2010). Model checking of linearizability of concurrent list implementations. In Touili, T., Cook, B., and Jackson, P. B., editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 465–479. Springer.

Chakraborty, S., Henzinger, T. A., Sezgin, A., and Vafeiadis, V. (2015). Aspect-oriented linearizability proofs. *Log. Methods Comput. Sci.*, 11(1).

da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., and Wheelhouse, M. J. (2011). A simple abstraction for complex concurrent indexes. In Lopes, C. V. and Fisher, K., editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming,*

Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, pages 845–864. ACM.

da Rocha Pinto, P., Dinsdale-Young, T., and Gardner, P. (2014). Tada: A logic for time and data abstraction. In Jones, R. E., editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer.

Dayan, N. and Idreos, S. (2018). Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In Das, G., Jermaine, C. M., and Bernstein, P. A., editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 505–520. ACM.

Delbianco, G. A., Sergey, I., Nanevski, A., and Banerjee, A. (2017). Concurrent data structures linked in time. In Müller, P., editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 8:1–8:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J., and Yang, H. (2013). Views: compositional reasoning for concurrent programs. In Giacobazzi, R. and Cousot, R., editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 287–300. ACM.

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. (2010). Concurrent abstract predicates. In D'Hondt, T., editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer.

Dodds, M., Haas, A., and Kirsch, C. M. (2015). A scalable, correct time-stamped stack. In Rajamani, S. K. and Walker, D., editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 233–246. ACM.

Dodds, M., Jagannathan, S., Parkinson, M. J., Svendsen, K., and Birkedal, L. (2016). Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2):4:1–4:72.

D'Osualdo, E., Farzan, A., Gardner, P., and Sutherland, J. (2019). Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *CoRR*, abs/1901.05750.

Dragoi, C., Gupta, A., and Henzinger, T. A. (2013). Automatic linearizability proofs of concurrent objects with cooperating updates. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 174–190. Springer.

Facebook (2020). RocksDB. https://rocksdb.org/. Last accessed on August 17, 2020.

Feldman, Y. M. Y., Enea, C., Morrison, A., Rinetzky, N., and Shoham, S. (2018). Order out of chaos: Proving linearizability using local views. In Schmid, U. and Widder, J., editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Feng, X., Ferreira, R., and Shao, Z. (2007). On the relationship between concurrent separation logic and assume-guarantee reasoning. In Nicola, R. D., editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer.

Filipovic, I., O'Hearn, P. W., Rinetzky, N., and Yang, H. (2009). Abstraction for concurrent objects. In Castagna, G., editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 252–266. Springer.

Frumin, D., Krebbers, R., and Birkedal, L. (2018). Reloc: A mechanised relational logic for fine-grained concurrency. In Dawar, A. and Grädel, E., editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 442–451. ACM.

Fu, M., Li, Y., Feng, X., Shao, Z., and Zhang, Y. (2010). Reasoning about optimistic concurrency using a program logic for history. In Gastin, P. and Laroussinie, F., editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 388–402. Springer.

Google (2020). LevelDB. https://github.com/google/leveldb. Last accessed on August 15, 2020.

Gotsman, A., Cook, B., Parkinson, M. J., and Vafeiadis, V. (2009). Proving that non-blocking algorithms don't block. In Shao, Z. and Pierce, B. C., editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 16–28. ACM.

Gu, R., Shao, Z., Kim, J., Wu, X. N., Koenig, J., Sjöberg, V., Chen, H., Costanzo, D., and Ramananandro, T. (2018). Certified concurrent abstraction layers. In Foster, J. S. and Grossman, D., editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661. ACM.

Harris, T. L. (2001). A pragmatic implementation of non-blocking linked-lists. In Welch, J. L., editor, *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer.

Hawblitzel, C., Petrank, E., Qadeer, S., and Tasiran, S. (2015). Automated and modular refinement reasoning for concurrent programs. In Kroening, D. and Pasareanu, C. S., editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 449–465. Springer.

Herlihy, M. and Shavit, N. (2008). *The art of multiprocessor programming*. Morgan Kaufmann.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

Heule, S., Leino, K. R. M., Müller, P., and Summers, A. J. (2013). Abstract read permissions: Fractional permissions without the fractions. In Giacobazzi, R., Berdine, J., and Mastroeni, I., editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 315–334. Springer.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Iris Team (2020). The Iris 3.3 documentation. https://plv.mpi-sws.org/iris/appendix-3.3.pdf. Last accessed on July 29, 2020.

Jonathan Ellis (2011). Leveled compaction in Apache Cassandra. https://www.datastax.com/blog/2011/10/leveled-compaction-apache-cassandra. Last accessed on August 15, 2020.

Jones, C. B. (1983). Specification and design of (parallel) programs. In Mason, R. E. A., editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 321–332. North-Holland/IFIP.

Jung, R., Krebbers, R., Birkedal, L., and Dreyer, D. (2016). Higher-order ghost state. In Garrigue, J., Keller, G., and Sumii, E., editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM.

Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. (2018). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20.

Jung, R., Lepigre, R., Parthasarathy, G., Rapoport, M., Timany, A., Dreyer, D., and Jacobs, B. (2020). The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32.

Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., and Dreyer, D. (2015). Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Rajamani, S. K. and Walker, D., editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM.

Khyzha, A., Dodds, M., Gotsman, A., and Parkinson, M. J. (2017). Proving linearizability using partial orders. In Yang, H., editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 639–667. Springer.

Krebbers, R., Jung, R., Bizjak, A., Jourdan, J., Dreyer, D., and Birkedal, L. (2017). The essence of higher-order concurrent separation logic. In Yang, H., editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer.

Krishna, S., Patel, N., Shasha, D. E., and Wies, T. (2020a). Verifying concurrent search structure templates. In Donaldson, A. F. and Torlak, E., editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 181–196. ACM.

Krishna, S., Shasha, D. E., and Wies, T. (2018). Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.*, 2(POPL):37:1–37:31.

Krishna, S., Summers, A. J., and Wies, T. (2020b). Local reasoning for global graph properties. In Müller, P., editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 308–335. Springer.

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In Clarke, E. M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer.

Leino, K. R. M. (2017). Modeling concurrency in dafny. In Bowen, J. P., Liu, Z., and Zhang, Z., editors, *Engineering Trustworthy Software Systems - Third International School, SETSS 2017, Chongqing, China, April 17-22, 2017, Tutorial Lectures*, volume 11174 of *Lecture Notes in Computer Science*, pages 115–142. Springer.

Leino, K. R. M. and Pit-Claudel, C. (2016). Trigger selection strategies to stabilize program verifiers. In Chaudhuri, S. and Farzan, A., editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer.

Levandoski, J. J., Lomet, D. B., and Sengupta, S. (2013). The bw-tree: A b-tree for new hardware platforms. In Jensen, C. S., Jermaine, C. M., and Zhou, X., editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 302–313. IEEE Computer Society.

Levandoski, J. J. and Sengupta, S. (2013). The bw-tree: A latch-free b-tree for log-structured flash storage. *IEEE Data Eng. Bull.*, 36(2):56–62.

Liang, H. and Feng, X. (2013). Modular verification of linearizability with non-fixed linearization points. In Boehm, H. and Flanagan, C., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 459–470. ACM.

Luo, C. and Carey, M. J. (2020). Lsm-based storage techniques: a survey. *VLDB J.*, 29(1):393–418.

Malecha, J. G., Morrisett, G., Shinnar, A., and Wisnesky, R. (2010). Toward a verified relational database management system. In Hermenegildo, M. V. and Palsberg, J., editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 237–248. ACM.

Meyer, R. and Wolff, S. (2019). Decoupling lock-free data structures from memory reclamation for static analysis. *Proc. ACM Program. Lang.*, 3(POPL):58:1–58:31.

Michael, M. and Scott, M. (1995). Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester.

Müller, P., Schwerhoff, M., and Summers, A. J. (2017). Viper: A verification infrastructure for permission-based reasoning. In Pretschner, A., Peled, D., and Hutzelmann, T., editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press.

Nanevski, A., Ley-Wild, R., Sergey, I., and Delbianco, G. A. (2014). Communicating state transition systems for fine-grained concurrent resources. In Shao, Z., editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer.

O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. (2001). Local reasoning about programs that alter data structures. In Fribourg, L., editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer.

O'Hearn, P. W., Rinetzky, N., Vechev, M. T., Yahav, E., and Yorsh, G. (2010). Verifying linearizability with hindsight. In Richa, A. W. and Guerraoui, R., editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 85–94. ACM.

O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. (1996). The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385.

Owicki, S. S. and Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285.

Piskac, R., Wies, T., and Zufferey, D. (2013). Automating separation logic using SMT. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789. Springer.

Piskac, R., Wies, T., and Zufferey, D. (2014). Grasshopper - complete heap verification with mixed specifications. In Ábrahám, E. and Havelund, K., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 124–139. Springer.

Raad, A., Villard, J., and Gardner, P. (2015). Colosl: Concurrent local subjective logic. In Vitek, J., editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 710–735. Springer.

Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. (2017). Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society.

Sergey, I., Nanevski, A., and Banerjee, A. (2015). Mechanized verification of fine-grained concurrent programs. In Grove, D. and Blackburn, S., editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 77–87. ACM.

Sergey, I., Nanevski, A., Banerjee, A., and Delbianco, G. A. (2016). Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In Visser, E. and Smaragdakis, Y., editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 92–110. ACM.

Severance, D. G. and Lohman, G. M. (1976). Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267.

Shasha, D. E. and Goodman, N. (1988). Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90.

Svendsen, K. and Birkedal, L. (2014). Impredicative concurrent abstract predicates. In Shao, Z., editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer.

Thonangi, R. and Yang, J. (2017). On log-structured merge for solid-state drives. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 683–694. IEEE Computer Society.

Vafeiadis, V. (2009). Shape-value abstraction for verifying linearizability. In Jones, N. D. and Müller-Olm, M., editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer.

Vafeiadis, V. and Parkinson, M. J. (2007). A marriage of rely/guarantee and separation logic. In Caires, L. and Vasconcelos, V. T., editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer.

Wu, X., Xu, Y., Shao, Z., and Jiang, S. (2015). Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In Lu, S. and Riedel, E., editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 71–82. USENIX Association.

Xiong, S., da Rocha Pinto, P., Ntzik, G., and Gardner, P. (2017). Abstract specifications for concurrent maps. In Yang, H., editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 964–990. Springer.

Zhu, H., Petri, G., and Jagannathan, S. (2015). Poling: SMT aided linearizability proofs. In Kroening, D. and Pasareanu, C. S., editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer.

Authors' Biographies

SIDDHARTH KRISHNA

Siddharth Krishna is a post-doctoral researcher at Microsoft Research Cambridge, UK. He did his PhD in logic and verification at New York University, where he had the good fortune to work with Nisarg, Dennis, and Thomas. He now works on parallel and distributed algorithms for large-scale mahine learning workloads. When he is not pleading with computers to do his bidding, he likes to watch comedy panel/news shows and play ultimate frisbee.

NISARG PATEL

Nisarg Patel is a PhD student at New York University's Department of Computer Science, where he works with Siddharth, Dennis and Thomas on automated verification of concurrent programs. His academic interests also include synthesis of controller programs for robots. Outside of computer science, he loves playing football and reading about history and politics.

DENNIS SHASHA

Dennis Shasha is a Julius Silver Professor of computer science at the Courant Institute of New York University and an Associate Director of NYU Wireless. In addition to his long fascination with concurrent algorithms, he works on meta-algorithms for machine learning to achieve guaranteed correctness rates; with biologists on pattern discovery for network inference; with physicists and financial people on algorithms for time series; on database tuning; and tree and graph matching.

Because he likes to type, he has written six books of puzzles about a mathematical detective named Dr. Ecco, a biography about great computer scientists, and a book about the future of computing. He has also written technical books about database tuning, biological pattern recognition, time series, DNA computing, resampling statistics, and causal inference in molecular networks.

He has written the puzzle column for various publications including *Scientific American*, *Dr. Dobb's Journal*, and currently the *Communications of the ACM*. He is a fellow of the ACM and an INRIA International Chair.

THOMAS WIES

Thomas Wies is an Associate Professor in computer science at the Courant Institute of New York University and a member of the Analysis of Computer Systems Group. His research interests are in

Programming Languages and Formal Methods with a focus on program analysis and verification, automated deduction, and correctness of concurrent software. He is recipient of an NSF CAREER Award and has won multiple best paper awards.